

Designing Efficient Kernels

For designers implementing a Vitis kernel there are various trade-offs available when working with the device memory (PLRAM, HBM and DDR) available on FPGA devices. The following is a checklist of best practices to use when designing AXI4 memory mapped interfaces for your application.

With throughput as the chief optimization goal, it is clear that accelerating the compute part of your application using the macro and micro-architecture optimizations is the first step but the time taken for transferring data to/from the kernel can also influence the application architecture with respect to throughput goals. Due to the high overhead for data transfer, it becomes important to think about overlapping the computation with the communication (data movement) that is present in your application.

For your given application:

- Decompose the kernel algorithm by building a pipeline of producer-consumer tasks, modeled using a Load, Compute, Store (LCS) coding pattern
 - All external I/O accesses must be in the Load and Store tasks.
 - There should be multiple Load or Store tasks if the kernel needs to read or write from different ports in parallel.
 - The Compute task(s) should only have scalars, array, streams or stream of blocks arguments.
 - Ensure that all three tasks (specified as functions) can be executed in overlapped fashion (enables task-level parallelism by the compiler).
 - Compute tasks can be further split up into smaller compute tasks which may contain further optimizations such as pipelining. The same rules as LCS apply for these smaller compute functions as well.
 - Always use local memory to pass data to/from the Compute task.
- Load and Store blocks are responsible for moving data between global memory and the Compute blocks as efficiently as possible.
 - On one end, they must read or write data through the streaming interface according to the (temporal) sequential order mandated by the Compute task inside the kernel
 - On the other end, they must read or write data through the memory-mapped interface according to the (spatial) arrangement order set by the software application

- Changing your mindset about data accesses is key to building a proper HW design with HLS
 - In SW, it is common to think about how the data is “accessed” (the algorithm *pulls* the data it needs).
 - In HW, it is more efficient in think of how data “flows” through the algorithm (the data is *pushed* to the algorithm)
 - In SW, you reason about array indices and “where” data is accessed
 - In HW, you reason about streams and “when” data is accessed
- Global memories have long access times (DRAM, HBM) and their bandwidth is limited (DRAM). To reduce the overhead of accessing global memory, the interface function needs to
 - Access sufficiently large contiguous blocks of data (to benefit from [bursting](#))
 - Accessing data sequentially leads to larger bursts (and higher data throughput efficiency) as compared to accessing random and/or out-of-order data (where burst analysis will fail)
 - Avoid redundant accesses (to preserve bandwidth)
- In many cases, the sequential order of data in and out of the Compute tasks is different from the arrangement order of data in global memory.
 - In this situation, optimizing the interface functions requires creating internal *caching* structures that gather enough data and organize it appropriately to minimize the overhead of global memory accesses while being able to satisfy the sequential order expected by the streaming interface
 - Example: [2D Convolution](#)
 - In order to simplify the data movement logic, the developer can also consider different ways of storing the data in memory. For instance, accessing data in DRAM in a row-major fashion can be very inefficient. Rather than implementing a dedicated data-mover in the kernel, it may be better to transpose the data in SW and store in column-major order instead which will greatly simplify HW access patterns.
- Maximize the port width of the interface, i.e., the bit-width of each AXI port by setting it to 512 bits (64 bytes).
 - Use [hls::vector](#) or [ap_\(u\)int<512>](#) as the data type of the port to infer maximal burst lengths. Usage of structs in the interface may result in poor burst performance.
 - Accessing the global memory is expensive and so accessing larger word sizes is more efficient.
 - Imagine the interface ports to be like pipes feeding data to your kernel. The wider the pipe, the more data that can be accessed and processed, and sent back.
 - Transfer large blocks of data from the global device memory. One large transfer is more efficient than several smaller transfers. The bandwidth is limited by the PCIe performance. Run the [DMA test](#) to measure PCIe® transfer effective max throughput. It is usually in the range of 10-17 GB/sec for reading and writing respectively.

- Memory resources include PLRAM (small size but fast access with the lowest latency), HBM (moderate size and access speed with some latency), and DRAM (large size but slow access with high latency).
 - Given the asynchronous nature of reads, distributed RAMs are ideal for fast buffers. You can use the read value immediately, rather than waiting for the next clock cycle. You can also use distributed RAM to create small ROMs. However, distributed ram is not suited for large memories, and you'll get better performance (and lower power consumption) for memories larger than about 128 bits using block RAM or UltraRAM.
- Decide on the optimal number of concurrent ports, i.e., the number of concurrent AXI (memory-mapped) ports
 - If the Load task needs to get multiple input data sets to feed to the Compute task, it can choose to use multiple interface ports to access this data in parallel.
 - However, the data needs to be stored in **different memory banks** or the accesses will be sequentialized. There is a maximum of 4 DDR banks on FPGAs while there are **32 HBM channels**.
 - When multiple processes are accessing the same memory port or memory bank, an **arbiter** will sequentialize these concurrent accesses to the same memory port or bank.
 - Setting the right burst length i.e., the maximum burst access length (in terms of the number of elements) for each AXI port.
 - Set the burst length equivalent to the maximum 4k bytes transfer. For example, using AXI data width of 512-bit (64 bytes), the burst length should be set to 64.
 - Transferring data in bursts hides the memory access latency and improves bandwidth usage and efficiency of the memory controller
 - Write application code in such a way to infer the maximal length bursts for both reads and writes to/from global memory
 - Setting the number of outstanding memory requests that an AXI port can sustain before stalling
 - Setting a reasonable number of **outstanding requests**, allows the system to submit multiple memory requests before stalling - this pipelining of requests allows the system to hide some of the memory latency at the cost of additional BRAM/URAM resources.

Vitis HLS Coding Styles

This chapter explains how various constructs of C and C++11/C++14 are synthesized into an FPGA hardware implementation, and discusses any restrictions with regard to standard C coding.

The coding examples in this guide are available on GitHub for use with the Vitis HLS release. You can clone the examples repository from GitHub by clicking the **Clone Examples** command from the Vitis HLS Welcome screen.

Note: To view the Welcome screen at any time, select **Help → Welcome**.

Unsupported C/C++ Constructs

While Vitis HLS supports a wide range of the C/C++ languages, some constructs are not synthesizable, or can result in errors further down the design flow. This section discusses areas in which coding changes must be made for the function to be synthesized and implemented in a device.

To be synthesized:

- The function must contain the entire functionality of the design.
- None of the functionality can be performed by system calls to the operating system.
- The C/C++ constructs must be of a fixed or bounded size.
- The implementation of those constructs must be unambiguous.

System Calls

System calls cannot be synthesized because they are actions that relate to performing some task upon the operating system in which the C/C++ program is running.

Vitis HLS ignores commonly-used system calls that display only data and that have no impact on the execution of the algorithm, such as `printf()` and `fprintf(stdout,)`. In general, calls to the system cannot be synthesized and should be removed from the function before synthesis. Other examples of such calls are `getc()`, `time()`, `sleep()`, all of which make calls to the operating system.

Vitis HLS defines the macro `__SYNTHESIS__` when synthesis is performed. This allows the `__SYNTHESIS__` macro to exclude non-synthesizable code from the design.

Note: Only use the `__SYNTHESIS__` macro in the code to be synthesized. Do not use this macro in the test bench, because it is not obeyed by C/C++ simulation or C/C++ RTL co-simulation.



CAUTION! You must not define or undefine the `__SYNTHESIS__` macro in code or with compiler options, otherwise compilation might fail.

In the following code example, the intermediate results from a sub-function are saved to a file on the hard drive. The macro `__SYNTHESIS__` is used to ensure the non-synthesizable files writes are ignored during synthesis.

```
#include "hier_func4.h"

int sumsub_func(dint_t *in1, dint_t *in2, dint_t *outSum, dint_t *outSub)
{
    *outSum = *in1 + *in2;
    *outSub = *in1 - *in2;
}

int shift_func(dint_t *in1, dint_t *in2, dout_t *outA, dout_t *outB)
{
    *outA = *in1 >> 1;
    *outB = *in2 >> 2;
}

void hier_func4(dint_t A, dint_t B, dout_t *C, dout_t *D)
{
    dint_t apb, amb;

    sumsub_func(&A,&B,&apb,&amb);
#ifndef __SYNTHESIS__
    FILE *fp1; // The following code is ignored for synthesis
    char filename[255];
    sprintf(filename,Out_apb_%03d.dat,apb);
    fp1=fopen(filename,w);
    fprintf(fp1, %d \n, apb);
    fclose(fp1);
#endif
    shift_func(&apb,&amb,C,D);
}
```

The `__SYNTHESIS__` macro is a convenient way to exclude non-synthesizable code without removing the code itself from the function. Using such a macro does mean that the code for simulation and the code for synthesis are now different.



CAUTION! If the `__SYNTHESIS__` macro is used to change the functionality of the C/C++ code, it can result in different results between C/C++ simulation and C/C++ synthesis. Errors in such code are inherently difficult to debug. Do not use the `__SYNTHESIS__` macro to change functionality.

Dynamic Memory Usage

Any system calls that manage memory allocation within the system, for example, `malloc()`, `alloc()`, and `free()`, are using resources that exist in the memory of the operating system and are created and released during runtime. To be able to synthesize a hardware implementation the design must be fully self-contained, specifying all required resources.

Memory allocation system calls must be removed from the design code before synthesis. Because dynamic memory operations are used to define the functionality of the design, they must be transformed into equivalent bounded representations. The following code example shows how a design using `malloc()` can be transformed into a synthesizable version and highlights two useful coding style techniques:

- The design does not use the `__SYNTHESIS__` macro.

The user-defined macro `NO_SYNTH` is used to select between the synthesizable and non-synthesizable versions. This ensures that the same code is simulated in C/C++ and synthesized in Vitis HLS.

- The pointers in the original design using `malloc()` do not need to be rewritten to work with fixed sized elements.

Fixed sized resources can be created and the existing pointer can simply be made to point to the fixed sized resource. This technique can prevent manual recoding of the existing design.

```
#include "malloc_removed.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t malloc_removed(din_t din[N], dsel_t width) {

#ifndef NO_SYNTH
    long long *out_accum = malloc (sizeof(long long));
    int* array_local = malloc (64 * sizeof(int));
#else
    long long _out_accum;
    long long *out_accum = &_out_accum;
    int _array_local[64];
    int* array_local = &_array_local[0];
#endif
    int i, j;

    LOOP_SHIFT:for (i=0;i<N-1; i++) {
        if (i<width)
            *(array_local+i)=din[i];
        else
            *(array_local+i)=din[i]>>2;
    }

    *out_accum=0;
    LOOP_ACCUM:for (j=0;j<N-1; j++) {

```

```
*out_accum += *(array_local+j);  
}  
  
return *out_accum;  
}
```

Because the coding changes here impact the functionality of the design, Xilinx does not recommend using the `__SYNTHESIS__` macro. Xilinx recommends that you perform the following steps:

1. Add the user-defined macro `NO_SYNTH` to the code and modify the code.
2. Enable macro `NO_SYNTH`, execute the C/C++ simulation, and save the results.
3. Disable the macro `NO_SYNTH`, and execute the C/C++ simulation to verify that the results are identical.
4. Perform synthesis with the user-defined macro disabled.

This methodology ensures that the updated code is validated with C/C++ simulation and that the identical code is then synthesized. As with restrictions on dynamic memory usage in C/C++, Vitis HLS does not support (for synthesis) C/C++ objects that are dynamically created or destroyed.

Pointer Limitations

General Pointer Casting

Vitis HLS does not support general pointer casting, but supports pointer casting between native C/C++ types.

Pointer Arrays

Vitis HLS supports pointer arrays for synthesis, provided that each pointer points to a scalar or an array of scalars. Arrays of pointers cannot point to additional pointers.

Function Pointers

Function pointers are not supported.

Note: Pointer to pointer is not supported.

Recursive Functions

Recursive functions cannot be synthesized. This applies to functions that can form endless recursion:

```
unsigned foo (unsigned n)
{
    if (n == 0 || n == 1) return 1;
    return (foo(n-2) + foo(n-1));
}
```

Vitis HLS also does not support tail recursion, in which there is a finite number of function calls.

```
unsigned foo (unsigned m, unsigned n)
{
    if (m == 0) return n;
    if (n == 0) return m;
    return foo(n, m%n);
}
```

In C++, templates can implement tail recursion and can then be used for synthesizable tail-recursive designs.

Note: Virtual Functions are not supported.

Standard Template Libraries

Many of the C++ Standard Template Libraries (STLs) contain function recursion and use dynamic memory allocation. For this reason, the STLs cannot be synthesized by Vitis HLS. The solution for STLs is to create a local function with identical functionality that does not feature recursion, dynamic memory allocation, or the dynamic creation and destruction of objects.

Note: Standard data types, such as `std::complex`, are supported for synthesis. However, the `std::complex<long double>` data type is not supported in Vitis HLS and should not be used.

Functions

The top-level function becomes the top level of the RTL design after synthesis. Sub-functions are synthesized into blocks in the RTL design.



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

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

Inlining Functions

Sub-functions can optionally be inlined to merge their logic with the logic of the surrounding function. While inlining functions can result in better optimizations, it can also increase runtime as more logic must be kept in memory and analyzed.



TIP: Vitis HLS can perform automatic inlining of small functions. To disable automatic inlining of a small function, set the `inline` directive to `off` for that function.

If a function is inlined, there is no report or separate RTL file for that function. The logic and loops of the sub-function are merged with the higher-level function in the hierarchy.

Impact of Coding Style

The primary impact of a coding style on functions is on the function arguments and interface.

If the arguments to a function are sized accurately, Vitis HLS can propagate this information through the design. There is no need to create arbitrary precision types for every variable. In the following example, two integers are multiplied, but only the bottom 24 bits are used for the result.

```
#include "ap_int.h"

ap_int<24> foo(int x, int y) {
    int tmp;

    tmp = (x * y);
    return tmp
}
```

When this code is synthesized, the result is a 32-bit multiplier with the output truncated to 24-bit.

If the inputs are correctly sized to 12-bit types (`int12`) as shown in the following code example, the final RTL uses a 24-bit multiplier.

```
#include "ap_int.h"
typedef ap_int<12> din_t;
typedef ap_int<24> dout_t;

dout_t func_sized(din_t x, din_t y) {
    int tmp;

    tmp = (x * y);
    return tmp
}
```

Using arbitrary precision types for the two function inputs is enough to ensure Vitis HLS creates a design using a 24-bit multiplier. The 12-bit types are propagated through the design. Xilinx recommends that you correctly size the arguments of all functions in the hierarchy.

In general, when variables are driven directly from the function interface, especially from the top-level function interface, they can prevent some optimizations from taking place. A typical case of this is when an input is used as the upper limit for a loop index.

C/C++ Builtin Functions

Vitis HLS supports the following C/C++ builtin functions:

- `__builtin_clz(unsigned int x)`: Returns the number of leading 0-bits in `x`, starting at the most significant bit position. If `x` is 0, the result is undefined.
- `__builtin_ctz(unsigned int x)`: Returns the number of trailing 0-bits in `x`, starting at the least significant bit position. If `x` is 0, the result is undefined.

The following example shows these functions may be used. This example returns the sum of the number of leading zeros in `in0` and trailing zeros in `in1`:

```
int foo (int in0, int in1) {  
    int ldz0 = __builtin_clz(in0);  
    int ldz1 = __builtin_ctz(in1);  
    return (ldz0 + ldz1);  
}
```

Loops

Loops provide a very intuitive and concise way of capturing the behavior of an algorithm and are used often in C/C++ code. Loops are very well supported by synthesis: loops can be pipelined, unrolled, partially unrolled, merged, and flattened.

The optimizations that unroll, partially unroll, flatten, and merge effectively make changes to the loop structure, as if the code was changed. These optimizations ensure limited coding changes are required when optimizing loops. Some optimizations can be applied only in certain conditions. Some coding changes might be required.



RECOMMENDED: Avoid use of global variables for loop index variables, as this can inhibit some optimizations.

Variable Loop Bounds

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

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

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

dout_t code028(din_t A[N], dsel_t width) {
    dout_t out_accum=0;
    dsel_t x;

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

    return out_accum;
}
```

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

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

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

Another issue with variable loop bounds is that the performance of the design is unknown. The two ways to overcome this issue are as follows:

- Use the [pragma HLS loop_tripcount](#) or [set_directive_loop_tripcount](#).
- Use an `assert` macro in the C/C++ code.

The `tripcount` directive allows a minimum and/or maximum `tripcount` to be specified for the loop. The `tripcount` is the number of loop iterations. If a maximum `tripcount` of 32 is applied to `LOOP_X` in the first example, the report is updated to the following:

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

The user-provided values for the `tripcount` directive are used only for reporting. The `tripcount` value allows Vitis HLS to report number in the report, allowing the reports from different solutions to be compared. To have this same loop-bound information used for synthesis, the C/C++ code must be updated.

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

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

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

```
@W [XFORM-503] Cannot unroll loop 'LOOP_X' in function 'code028': cannot
completely
unroll a loop with a variable trip count.
```

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



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

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

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

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

dout_t loop_max_bounds(din_t A[N], dsel_t width) {
    dout_t out_accum=0;
```

```
dsel_t x;

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

return out_accum;
}
```

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

Loop Pipelining

When pipelining loops, the optimal balance between area and performance is typically found by pipelining the innermost loop. This is also results in the fastest runtime. The following code example demonstrates the trade-offs when pipelining loops and functions.

```
#include "loop_pipeline.h"

dout_t loop_pipeline(din_t A[N]) {

    int i,j;
    static dout_t acc;

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

    return acc;
}
```

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



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

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

If the top-level function is pipelined, both loops must be unrolled: 400 multipliers and 400 arrays accessed must now be scheduled. It is very unlikely that Vitis HLS will produce a design with 400 multiplications because in most designs, data dependencies often prevent maximal parallelism, for example, even if a dual-port RAM is used for $A[N]$, the design can only access two values of $A[N]$ in any clock cycle.

The concept to appreciate when selecting at which level of the hierarchy to pipeline is to understand that pipelining the innermost loop gives the smallest hardware with generally acceptable throughput for most applications. Pipelining the upper levels of the hierarchy unrolls all sub-loops and can create many more operations to schedule (which could impact runtime and memory capacity), but typically gives the highest performance design in terms of throughput and latency.

To summarize the above options:

- Pipeline `LOOP_J`

Latency is approximately 400 cycles (20x20) and requires less than 100 LUTs and registers (the I/O control and FSM are always present).

- Pipeline `LOOP_I`

Latency is approximately 20 cycles but requires a few hundred LUTs and registers. About 20 times the logic as first option, minus any logic optimizations that can be made.

- Pipeline function `loop_pipeline`

Latency is approximately 10 (20 dual-port accesses) but requires thousands of LUTs and registers (about 400 times the logic of the first option minus any optimizations that can be made).

Imperfect Nested Loops

When the inner loop of a loop hierarchy is pipelined, Vitis HLS flattens the nested loops to reduce latency and improve overall throughput by removing any cycles caused by loop transitioning (the checks performed on the loop index when entering and exiting loops). Such checks can result in a clock delay when transitioning from one loop to the next (entry and/or exit).

Imperfect loop nests, or the inability to flatten them, results in additional clock cycles to enter and exit the loops. When the design contains nested loops, analyze the results to ensure as many nested loops as possible have been flattened: review the log file or look in the synthesis report for cases, as shown in [Loop Pipelining](#), where the loop labels have been merged (`LOOP_I` and `LOOP_J` are now reported as `LOOP_I_LOOP_J`).

Loop Parallelism

Vitis HLS schedules logic and functions early as possible to reduce latency while keeping the estimated clock period below the user-specified period. To perform this, it schedules as many logic operations and functions as possible in parallel. It does not schedule loops to execute in parallel.

If the following code example is synthesized, loop `SUM_X` is scheduled and then loop `SUM_Y` is scheduled: even though loop `SUM_Y` does not need to wait for loop `SUM_X` to complete before it can begin its operation, it is scheduled after `SUM_X`.

```
#include "loop_sequential.h"

void loop_sequential(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
    dsel_t xlimit, dsel_t ylimit) {

    dout_t X_accum=0;
    dout_t Y_accum=0;
    int i,j;

    SUM_X:for (i=0;i<xlimit; i++) {
        X_accum += A[i];
        X[i] = X_accum;
    }

    SUM_Y:for (i=0;i<ylimit; i++) {
        Y_accum += B[i];
        Y[i] = Y_accum;
    }
}
```

Because the loops have different bounds (`xlimit` and `ylimit`), they cannot be merged. By placing the loops in separate functions, as shown in the following code example, the identical functionality can be achieved and both loops (inside the functions), can be scheduled in parallel.

```
#include "loop_functions.h"

void sub_func(din_t I[N], dout_t O[N], dsel_t limit) {
    int i;
    dout_t accum=0;

    SUM:for (i=0;i<limit; i++) {
        accum += I[i];
        O[i] = accum;
    }
}

void loop_functions(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
    dsel_t xlimit, dsel_t ylimit) {
    sub_func(A,X,xlimit);
    sub_func(B,Y,ylimit);
}
```

If the previous example is synthesized, the latency is half the latency of the sequential loops example because the loops (as functions) can now execute in parallel.

The `dataflow` optimization could also be used in the sequential loops example. The principle of capturing loops in functions to exploit parallelism is presented here for cases in which `dataflow` optimization cannot be used. For example, in a larger example, `dataflow` optimization is applied to all loops and functions at the top-level and memories placed between every top-level loop and function.

Loop Dependencies

Loop dependencies are data dependencies that prevent optimization of loops, typically pipelining. They can be within a single iteration of a loop and or between different iteration of a loop.

The easiest way to understand loop dependencies is to examine an extreme example. In the following example, the result of the loop is used as the loop continuation or exit condition. Each iteration of the loop must finish before the next can start.

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

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

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

Unrolling Loops in C++ Classes

When loops are used in C++ classes, care should be taken to ensure the loop induction variable is not a data member of the class as this prevents the loop from being unrolled.

In this example, loop induction variable `k` is a member of class `foo_class`.

```
template <typename T0, typename T1, typename T2, typename T3, int N>
class foo_class {
private:
    pe_mac<T0, T1, T2> mac;
public:
    T0 areg;
    T0 breg;
```

```

T2 mreg;
T1 preg;
    T0 shift[N];
int k;           // Class Member
    T0 shift_output;
void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
{
Function_label0:;
#pragma HLS inline off
SRL:for (k = N-1; k >= 0; --k) {
#pragma HLS unroll // Loop will fail UNROLL
if (k > 0)
shift[k] = shift[k-1];
else
shift[k] = data;
}

    *dataOut = shift_output;
    shift_output = shift[N-1];
}

*pcout = mac.exec1(shift[4*col], coeff, pcin);
};

```

For Vitis HLS to be able to unroll the loop as specified by the UNROLL pragma directive, the code should be rewritten to remove k as a class member.

```

template <typename T0, typename T1, typename T2, typename T3, int N>
class foo_class {
private:
    pe_mac<T0, T1, T2> mac;
public:
    T0 areg;
    T0 breg;
    T2 mreg;
    T1 preg;
        T0 shift[N];
        T0 shift_output;
void exec(T1 *pcout, T0 *dataOut, T1 pcin, T3 coeff, T0 data, int col)
{
Function_label0:;
    int k;           // Local variable
#pragma HLS inline off
SRL:for (k = N-1; k >= 0; --k) {
#pragma HLS unroll // Loop will unroll
if (k > 0)
shift[k] = shift[k-1];
else
shift[k] = data;
}

    *dataOut = shift_output;
    shift_output = shift[N-1];
}

*pcout = mac.exec1(shift[4*col], coeff, pcin);
};

```

Arrays

Before discussing how the coding style can impact the implementation of arrays after synthesis it is worthwhile discussing a situation where arrays can introduce issues even before synthesis is performed, for example, during C/C++ simulation.

If you specify a very large array, it might cause C/C++ simulation to run out of memory and fail, as shown in the following example:

```
#include "ap_int.h"

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

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

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

This might mean the design runs out of memory when running and certain issues might make this issue more likely:

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

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

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

In some cases, the machine may not have enough available memory and increasing the stack size does not help.

A solution is to use dynamic memory allocation for simulation but a fixed sized array for synthesis, as shown in the next example. This means that the memory required for this is allocated on the heap, managed by the OS, and which can use local disk space to grow.

A change such as this to the code is not ideal, because the code simulated and the code synthesized are now different, but this might sometimes be the only way to move the design process forward. If this is done, be sure that the C/C++ test bench covers all aspects of accessing the array. The RTL simulation performed by `cosim_design` will verify that the memory accesses are correct.

```
#include "ap_int.h"

    int i, acc;
#endif __SYNTHESIS__
    // Use an arbitrary precision type & array for synthesis
    ap_int<32> la0[10000000], la1[10000000];
#else
    // Use an arbitrary precision type & dynamic memory for simulation
    ap_int<int32> *la0 = malloc(10000000 * sizeof(ap_int<32>));
    ap_int<int32> *la1 = malloc(10000000 * sizeof(ap_int<32>));
#endif
    for (i=0 ; i < 10000000; i++) {
        acc = acc + la0[i] + la1[i];
    }
```

Note: Only use the `__SYNTHESIS__` macro in the code to be synthesized. Do not use this macro in the test bench, because it is not obeyed by C/C++ simulation or C/C++ RTL co-simulation.

Arrays are typically implemented as a memory (RAM, ROM or FIFO) after synthesis. Arrays on the top-level function interface are synthesized as RTL ports that access a memory outside. Internal to the design, arrays sized less than 1024 will be synthesized as FIFO. Arrays sized greater than 1024 will be synthesized into block RAM, LUTRAM, and UltraRAM depending on the optimization settings.

Like loops, arrays are an intuitive coding construct and so they are often found in C/C++ programs. Also like loops, Vitis HLS includes optimizations and directives that can be applied to optimize their implementation in RTL without any need to modify the code.

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

- Array accesses can often create bottlenecks to performance. When implemented as a memory, the number of memory ports limits access to the data.
- Some care must be taken to ensure arrays that only require read accesses are implemented as ROMs in the RTL.

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

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

Array Accesses and Performance

The following code example shows a case in which accesses to an array can limit performance in the final RTL design. In this example, there are three accesses to the array `mem[N]` to create a summed result.

```
#include "array_mem_bottleneck.h"

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

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

    return sum;
}
```

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

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

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

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

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

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



CAUTION! Arrays implemented as memory or memory ports can often become bottlenecks to performance.

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

```
#include "array_mem_perform.h"

dout_t array_mem_perform(din_t mem[N]) {

    din_t tmp0, tmp1, tmp2;
    dout_t sum=0;
    int i;

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

    return sum;
}
```

Vitis HLS includes optimization directives for changing how arrays are implemented and accessed. It is typically the case that directives can be used, and changes to the code are not required. Arrays can be partitioned into blocks or into their individual elements. In some cases, Vitis HLS partitions arrays into individual elements. This is controllable using the configuration settings for auto-partitioning.

When an array is partitioned into multiple blocks, the single array is implemented as multiple RTL RAM blocks. When partitioned into elements, each element is implemented as a register in the RTL. In both cases, partitioning allows more elements to be accessed in parallel and can help with performance; the design trade-off is between performance and the number of RAMs or registers required to achieve it.

FIFO Accesses

A special case of arrays accesses are when arrays are implemented as FIFOs. This is often the case when dataflow optimization is used.

Accesses to a FIFO must be in sequential order starting from location zero. In addition, if an array is read in multiple locations, the code must strictly enforce the order of the FIFO accesses. It is often the case that arrays with multiple fanout cannot be implemented as FIFOs without additional code to enforce the order of the accesses.

Arrays on the Interface

In the Vivado IP flow Vitis HLS synthesizes arrays into memory elements by default. When you use an array as an argument to the top-level function, Vitis HLS assumes one of the following:

- Memory is off-chip.
Vitis HLS synthesizes interface ports to access the memory.
- Memory is standard block RAM with a latency of 1.
The data is ready one clock cycle after the address is supplied.

To configure how Vitis HLS creates these ports:

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



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

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

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

Array Interfaces

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

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

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

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

```
#include "array_RAM.h"

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

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

A single-port RAM interface is used because the `for`-loop ensures that only one element can be read and written in each clock cycle. There is no advantage in using a dual-port RAM interface.

If the `for`-loop is unrolled, Vitis HLS uses a dual-port RAM. Doing so allows multiple elements to be read at the same time and improves the initiation interval. The type of RAM interface can be explicitly set by applying the INTERFACE pragma or directive, and setting the `storage_type`.

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

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

FIFO Interfaces

Vitis HLS allows array arguments to be implemented as FIFO ports in the RTL. If a FIFO port is to be used, be sure that the accesses to and from the array are sequential. Vitis HLS conservatively tries to determine whether the accesses are sequential.

Table 13: Vitis HLS Analysis of Sequential Access

Accesses Sequential?	Vitis HLS Action
Yes	Implements the FIFO port.
No	<ol style="list-style-type: none"> Issues an error message. Halts synthesis.
Indeterminate	<ol style="list-style-type: none"> Issues a warning. Implements the FIFO port.

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

The following code example shows a case in which Vitis HLS cannot determine whether the accesses are sequential. In this example, both `d_i` and `d_o` are specified to be implemented with a FIFO interface during synthesis.

```
#include "array_FIFO.h"

void array_FIFO (dout_t d_o[4], din_t d_i[4], didx_t idx[4]) {
    int i;
#pragma HLS INTERFACE mode=ap_fifo port=d_i
#pragma HLS INTERFACE mode=ap_fifo port=d_o
    // Breaks FIFO interface d_o[3] = d_i[2];
    For_Loop: for (i=0;i<4;i++) {
        d_o[i] = d_i[idx[i]];
    }
}
```

In this case, the behavior of variable `idx` determines whether or not a FIFO interface can be successfully created.

- If `idx` is incremented sequentially, a FIFO interface can be created.
- If random values are used for `idx`, a FIFO interface fails when implemented in RTL.

Because this interface might not work, Vitis HLS issues a message during synthesis and creates a FIFO interface.

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

If you remove `//Breaks FIFO interface` comment in the example above, leaving the remaining portion of the line uncommented, `d_o[3] = d_i[2];`, Vitis HLS can determine that the accesses to the arrays are not sequential, and it halts with an error message if a FIFO interface is specified.

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

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

- The array must be written and read in only one loop or function. This can be transformed into a point-to-point connection that matches the characteristics of FIFO links.
- The array reads must be in the same order as the array write. Because random access is not supported for FIFO channels, the array must be used in the program following first in, first out semantics.
- The index used to read and write from the FIFO must be analyzable at compile time. Array addressing based on runtime computations cannot be analyzed for FIFO semantics and prevent the tool from converting an array into a FIFO.

Code changes are generally not required to implement or optimize arrays in the top-level interface. The only time arrays on the interface may need coding changes is when the array is part of a struct.

Array Initialization



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

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

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

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

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

In addition, if the variable has the `static` qualifier, Vitis HLS initializes the variable in the RTL design and in the FPGA bitstream. This removes the need for multiple clock cycles to initialize the memory and ensures that initializing large memories is not an operational overhead.

The RTL configuration command can specify if static variables return to their initial state after a reset is applied (not the default). If a memory is to be returned to its initial state after a reset operation, this incurs an operational overhead and requires multiple cycles to reset the values. Each value must be written into each memory address.

Implementing ROMs

Vitis HLS does not require that an array be specified with the `static` qualifier to synthesize a memory or the `const` qualifier to infer that the memory should be a ROM. Vitis HLS analyzes the design and attempts to create the most optimal hardware.



IMPORTANT! Xilinx highly recommends using the `static` qualifier for arrays that are intended to be memories. As noted in [Array Initialization](#), a `static` type behaves in an almost identical manner as a memory in RTL.

The `const` qualifier is also recommended when arrays are only read, because Vitis HLS cannot always infer that a ROM should be used by analysis of the design. The general rule for the automatic inference of a ROM is that a local (non-global), `static` array is written to before being read. The following practices in the code can help infer a ROM:

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

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

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

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

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



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

Data Types



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

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

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

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

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

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

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

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

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

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

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

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

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



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

Arbitrary Precision (AP) Data Types

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

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

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



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

AP Example

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

Using arbitrary precision data types in this design, you can specify the exact bit-sizes needed in your code prior to synthesis, simulate the updated code, and verify the results prior to synthesis.

Advantages of AP Data Types



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

The following code performs some basic arithmetic operations:

```
#include "types.h"

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

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

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

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

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

The design gives the following results after synthesis:

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

+ Latency (clock cycles):
  * Summary:
    +-----+-----+-----+-----+
    | Latency | Interval| Pipeline|
    | min | max | min | max | Type |
    +-----+-----+-----+-----+
    | 66 | 66 | 67 | 67 | none |
    +-----+-----+-----+-----+
  * Summary:
+-----+-----+-----+-----+-----+
| Name | BRAM_18K| DSP48E| FF | LUT |
+-----+-----+-----+-----+
| Expression | - | - | 0 | 17 |
| FIFO | - | - | - | - |
| Instance | - | 1 | 17920 | 17152 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | - |
| Register | - | - | 7 | - |
+-----+-----+-----+-----+
| Total | 0 | 1 | 17927 | 17169 |
+-----+-----+-----+-----+
| Available | 650 | 600 | 202800 | 101400 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | ~0 | 8 | 16 |
+-----+-----+-----+-----+
```

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

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

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

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

+ Latency (clock cycles):
  * Summary:
  +-----+-----+-----+
  | Latency | Interval | Pipeline|
  | min | max | min | max | Type |
  +-----+-----+-----+
  | 35 | 35 | 36 | 36 | none |
  +-----+-----+-----+

* Summary:
+-----+-----+-----+-----+-----+
| Name | BRAM_18K | DSP48E1 | FF | LUT |
+-----+-----+-----+-----+-----+
| Expression | - | - | 0 | 13 |
| FIFO | - | - | - | - |
| Instance | - | 1 | 4764 | 4560 |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | - |
| Register | - | - | 6 | - |
+-----+-----+-----+-----+
| Total | 0 | 1 | 4770 | 4573 |
+-----+-----+-----+-----+
| Available | 650 | 600 | 202800 | 101400 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | ~0 | 2 | 4 |
+-----+-----+-----+-----+
```

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

Overview of Arbitrary Precision Integer Data Types

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

Table 14: Arbitrary Precision Data Types

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

The header files which define the arbitrary precision types are also provided with Vitis HLS as a standalone package with the rights to use them in your own source code. The package, `xilinx_hls_lib_<release_number>.tgz` is provided in the include directory in the Vitis HLS installation area. The package does not include the C arbitrary precision types defined in `ap_cint.h`. These types cannot be used with standard C compilers.

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

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

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

```
#include "ap_int.h"

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

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



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

The following is an example of overriding `AP_INT_MAX_W`:

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

ap_int<4096> very_wide_var;
```

Overview of Arbitrary Precision Fixed-Point Data Types

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

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

When performing calculations where the variables have different number of bits or different precision, the binary point is automatically aligned.

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

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

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

Table 15: Fixed-Point Identifier Summary

Identifier	Description
W	Word length in bits
I	<p>The number of bits used to represent the integer value, that is, the number of integer bits to the <i>left</i> of the binary point. When this value is negative, it represents the number of <i>implicit</i> sign bits (for signed representation), or the number of <i>implicit</i> zero bits (for unsigned representation) to the <i>right</i> of the binary point. For example,</p> <pre>ap_fixed<2, 0> a = -0.5; // a can be -0.5, ap_ufixed<1, 0> x = 0.5; // 1-bit representation. x can be 0 or 0.5 ap_ufixed<1, -1> y = 0.25; // 1-bit representation. y can be 0 or 0.25 const ap_fixed<1, -7> z = 1.0/256; // 1-bit representation for z = 2^-8</pre>

Table 15: Fixed-Point Identifier Summary (cont'd)

Identifier	Description	
Q	Quantization mode: This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result.	
	ap_fixed Types	Description
	AP_RND	Round to plus infinity
	AP_RND_ZERO	Round to zero
	AP_RND_MIN_INF	Round to minus infinity
	AP_RND_INF	Round to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity (default)
	AP_TRN_ZERO	Truncation to zero
O	Overflow mode: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result.	
	ap_fixed Types	Description
	AP_SAT ¹	Saturation
	AP_SAT_ZERO ¹	Saturation to zero
	AP_SAT_SYM ¹	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	This defines the number of saturation bits in overflow wrap modes.	

Notes:

1. Using the AP_SAT* modes can result in higher resource usage as extra logic will be needed to perform saturation and this extra cost can be as high as 20% additional LUT usage.

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



IMPORTANT! ROM Synthesis can take a long time when using APFixed:. Changing it to int results in a quicker synthesis. For example:

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

Can be changed to:

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

Standard Types

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

```
#include "types_standard.h"

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

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

}
```

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

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

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

#define N 9

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

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

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

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

- The multiplier used to calculate result `out1` is a 24-bit multiplier. An 8-bit `char` type multiplied by a 16-bit `short` type requires a 24-bit multiplier. The result is sign-extended to 32-bit to match the output port width.
- The adder used for `out2` is 8-bit. Because the output is an 8-bit `unsigned char` type, only the bottom 8-bits of `inB` (a 16-bit `short`) are added to 8-bit `char` type `inA`.

- For output `out3` (32-bit exact width type), 8-bit `char` type `inA` is sign-extended to 32-bit value and a 32-bit division operation is performed with the 32-bit (`int` type) `inC` input.
- A 64-bit modulus operation is performed using the 64-bit `long long` type `inD` and 8-bit `char` type `inA` sign-extended to 64-bit, to create a 64-bit output result `out4`.

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

Floats and Doubles

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

- Single-precision 32-bit
 - 24-bit fraction
 - 8-bit exponent
- Double-precision 64-bit
 - 53-bit fraction
 - 11-bit exponent



RECOMMENDED: When using floating-point data types, Xilinx highly recommends that you review *Floating-Point Design with Vivado HLS (XAPP599)*.

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

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

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

#define N 9

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

typedef double dout_1;
typedef double dout_2;
```

```

typedef double dout_3;
typedef float dout_4;

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

```

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

```

#include "types_float_double.h"

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

    // Basic arithmetic & math.h sqrtf()
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = sqrtf(inD);

}

```

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

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

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

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

```

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

```

The above code results in the following hardware:

```

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

```

Using a `sqrtf()` function:

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

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

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

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



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

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

Floating-Point Accumulator and MAC

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

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

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

- Low-precision accumulation is suitable for high-throughput low-precision floating point accumulation and multiply-accumulation, this mode is only available in non-Versal devices.
 - It uses an integer accumulator with a pre-scaler and a post-scaler (to convert input and output to single-precision or double-precision floating point).
 - It uses a 60 bit and 100 bit accumulator for single and double precision inputs respectively.
 - It can cause cosim mismatches due to insufficient precision with respect to C++ simulation
 - It can always be pipelined with an `II=1` without source code changes

- It uses approximately 3X the resources of standard-precision floating point accumulation, which achieves an II that is typically between 3 and 5, depending on clock frequency and target device.

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

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

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



TIP: To achieve the same results as in prior releases, use the following configuration:

```
config_op facc -impl auto -precision low
```

Composite Data Types

HLS supports composite data types for synthesis:

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

Structs

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



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

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



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

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

```
config_unroll -tripcount_threshold 16
```

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

Structs on the Interface

Structs on the interface are aggregated by Vitis HLS by default; combining all of the elements of a struct into a single wide vector. This allows all members of the struct to be read and written-to simultaneously.



IMPORTANT! Structs on the interface are aggregated by default but can be disaggregated using the **DISAGGREGATE** pragma or directive. Structs on the interface also prevent **Automatic Port Width Resizing** and must be coded as separate elements to enable that feature.

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

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

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

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

```
typedef struct {
    unsigned short varA;
    unsigned char varB[4];
} data_t;

data_t struct_port(data_t i_val, data_t *i_pt, data_t *o_pt);
```

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



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

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

Struct Padding and Alignment

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

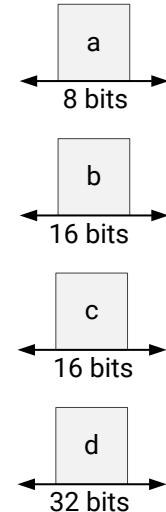
- **Disaggregate:** By default, structs in the code as internal variables are disaggregated into individual elements. The number and type of elements created are determined by the contents of the struct itself. Vitis HLS will decide whether a struct will be disaggregated or not based on certain optimization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



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

Figure 51: Disaggregated Struct

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



X24681-100520

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



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

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



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

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

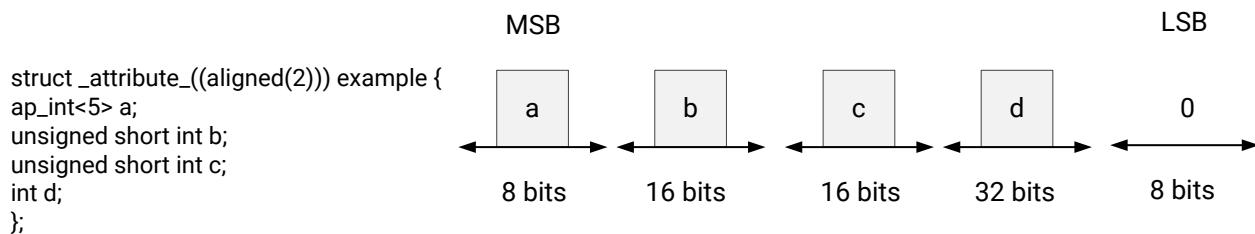
Data types in struct with custom data widths, such as `ap_int`, are allocated with sizes which are powers of 2. Vitis HLS adds padding bits for aligning the size of the data type to a power of 2.

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

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

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

Figure 52: Aligned Struct Implementation



X24682-102220

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

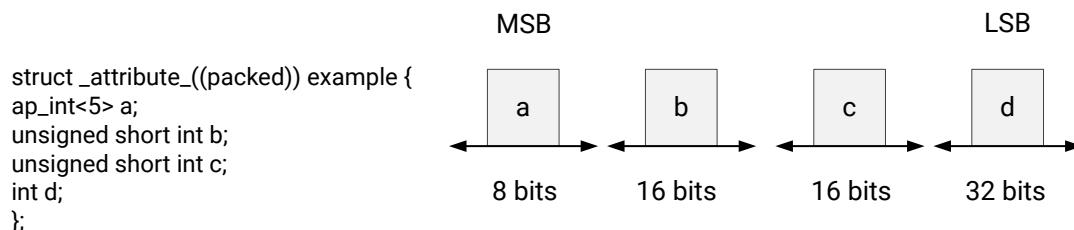
```
struct data_t {
    short varA;
    int varB;
    short varC;
};
```

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

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

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

Figure 53: Packed Struct Implementation



X24680-102220



TIP: This can also be achieved using the `compact-bit` option of the `AGGREGATE` pragma or directive.

Enumerated Types

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

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

```
#include <stdio.h>

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

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

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

typedef signed int mad_fixed_t;

typedef struct mad_header {
    enum mad_layer layer;
    enum mad_mode mode;
    int mode_extension;
    enum mad_emphasis emphasis;

    unsigned long long bitrate;
    unsigned int samplerate;
}
```

```

unsigned short crc_check;
unsigned short crc_target;

int flags;
int private_bits;

} header_t;

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

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

void types_composite(frame_t *frame);

```

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

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

```

#include "types_composite.h"

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

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

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

```

Unions

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

```
#include "types_union.h"

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

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

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

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

Vitis HLS does *not* support the following:

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

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

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

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

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

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

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

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

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

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

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

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

Type Qualifiers

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

Volatile

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

Note: Accesses to/from volatile variables is preserved. Currently the volatile per object is supported. This means:

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

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

Statics

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

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

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

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

Const

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

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

ROM Optimization

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

```
#include "array_ROM.h"

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

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

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

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

Global Variables

Global variables can be freely used in the code and are fully synthesizable. By default, global variables are not exposed as ports on the RTL interface.

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

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

```
din_t Ain[N];
din_t Aint[N];
dout_t Aout[N/2];

void types_global(din1_t idx) {
    int i,lidx;

    // Move elements in the input array
    for (i=0; i<N; ++i) {
        lidx=i;
        if(lidx+idx>N-1)
            lidx=i-N;
        Aint[lidx] = Ain[lidx+idx] + Ain[lidx];
    }

    // Sum to half the elements
```

```

for (i=0; i<(N/2); i++) {
    Aout[i] = (Aint[i] + Aint[i+1])/2;
}
    
```

By default, after synthesis, the only port on the RTL design is port `idx`. Global variables are not exposed as RTL ports by default. In the default case:

- Array `Ain` is an internal RAM that is *read from*.
- Array `Aout` is an internal RAM that is *written to*.

Pointers

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

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

Note: Pointer to pointer is not supported.

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

```

#include "pointer_multi.h"

dout_t pointer_multi (sel_t sel, din_t pos) {
    static const dout_t a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
    static const dout_t b[8] = {8, 7, 6, 5, 4, 3, 2, 1};

    dout_t* ptr;
    if (sel)
        ptr = a;
    else
        ptr = b;

    return ptr[pos];
}
    
```

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

```

#include "pointer_double.h"

data_t sub(data_t ptr[10], data_t size, data_t**flagPtr)
{
    data_t x, i;
    
```

```

x = 0;
// Sum x if AND of local index and pointer to pointer index is true
for(i=0; i<size; ++i)
    if (**flagPtr & i)
        x += *(ptr+i);
return x;
}

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

    ptrFlag = flag;

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

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

```

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

```

#include "pointer_array.h"

data_t A[N][10];

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

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

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

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

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

    return sum1;
}

```

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

```
#define N 1024

typedef int data_t;
typedef char dint_t;

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

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

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

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

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

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

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

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

Pointers on the Interface

Pointers can be used as arguments to the top-level function. It is important to understand how pointers are implemented during synthesis, because they can sometimes cause issues in achieving the desired RTL interface and design after synthesis.

Basic Pointers

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



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

```
#include "pointer_basic.h"

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

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

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

```
#include "pointer_basic.h"

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

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

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

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

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

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

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

Pointer Arithmetic

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

```
#include "pointer_arith.h"

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

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

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

```
#include "pointer_arith.h"

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

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

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

    // Save the results to a file
    fp=fopen(result.dat,w);
    printf( Din Dout\n, i, d );
    for (i=0;i<4;i++) {
        fprintf(fp, %d \n, d[i]);
        printf( %d %d\n, ref[i], d[i]);
    }
    fclose(fp);

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

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

When simulated, this results in the following output:

```
Din Dout
 0   1
 1   3
 2   6
 3  10
Test passed!
```

The pointer arithmetic does not access the pointer data in sequence. Wire, handshake, or FIFO interfaces have no way of accessing data out of order:

- A wire interface reads data when the design is ready to consume the data or write the data when the data is ready.
- Handshake and FIFO interfaces read and write when the control signals permit the operation to proceed.

In both cases, the data must arrive (and is written) in order, starting from element zero. In the Interface with Pointer Arithmetic example, the code states the first data value read is from index 1 (i starts at 0, $0+1=1$). This is the second element from array $d[5]$ in the test bench.

When this is implemented in hardware, some form of data indexing is required. Vitis HLS does not support this with wire, handshake, or FIFO interfaces.

Alternatively, the code must be modified with an array on the interface instead of a pointer, as in the following example. This can be implemented in synthesis with a RAM (`ap_memory`) interface. This interface can index the data with an address and can perform out-of-order, or non-sequential, accesses.

Wire, handshake, or FIFO interfaces can be used only on streaming data. It cannot be used with pointer arithmetic (unless it indexes the data starting at zero and then proceeds sequentially).

```
#include "array_arith.h"

void array_arith (dio_t d[5]) {
    static int acc = 0;
    int i;

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

Multi-Access Pointers on the Interface



IMPORTANT! Although multi-access pointers are supported on the interface, it is strongly recommended that you implement the required behavior using the `hls::stream` class instead of multi-access pointers to avoid some of the difficulties discussed below. Details on the `hls::stream` class can be found in [HLS Stream Library](#).

Designs that use pointers in the argument list of the top-level function (on the interface) need special consideration when multiple accesses are performed using pointers. Multiple accesses occur when a pointer is *read from* or *written to* multiple times in the same function.

Using pointers which are accessed multiple times can introduce unexpected behavior after synthesis. In the following "bad" example pointer `d_i` is read four times and pointer `d_o` is written to twice: the pointers perform multiple accesses.

```
#include "pointer_stream_bad.h"

void pointer_stream_bad ( dout_t *d_o, din_t *d_i) {
    din_t acc = 0;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```

After synthesis this code will result in an RTL design which reads the input port once and writes to the output port once. As with any standard C/C++ compiler, Vitis HLS will optimize away the redundant pointer accesses. The test bench to verify this design is shown in the following code example:

```
#include "pointer_stream_bad.h"
int main () {
    din_t d_i;
    dout_t d_o;
    int retval=0;
    FILE *fp;

    // Open a file for the output results
    fp=fopen(result.dat,w);

    // Call the function to operate on the data
    for (d_i=0;d_i<4;d_i++) {
        pointer_stream_bad(&d_o,&d_i);
        fprintf(fp, %d %d\n, d_i, d_o);
    }
    fclose(fp);

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

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

To implement the code as written, with the “anticipated” 4 reads on `d_i` and 2 writes to the `d_o`, the pointers must be specified as `volatile` as shown in the “pointer_stream_better” example.

```
#include "pointer_stream_better.h"

void pointer_stream_better ( volatile dout_t *d_o, volatile din_t *d_i) {
    din_t acc = 0;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```

To support multi-access pointers on the interface you should take the following steps:

- Validate the C/C++ before synthesis to confirm the intent and that the C/C++ model is correct.
- The pointer argument must have the number of accesses on the port interface specified when verifying the RTL using co-simulation within Vitis HLS.

Even this “better” C/C++ code is problematic. Indeed, using a test bench, there is no way to supply anything but a single value to `d_i` or verify any write to `d_o` other than the final write. Implement the required behavior using the `hls::stream` class instead of multi-access pointers.

Understanding Volatile Data

The code in [Multi-Access Pointers on the Interface](#) is written with *intent* that input pointer `d_i` and output pointer `d_o` are implemented in RTL as FIFO (or handshake) interfaces to ensure that:

- Upstream producer modules supply new data each time a read is performed on RTL port `d_i`.
- Downstream consumer modules accept new data each time there is a write to RTL port `d_o`.

When this code is compiled by standard C/C++ compilers, the multiple accesses to each pointer is reduced to a single access. As far as the compiler is concerned, there is no indication that the data on `d_i` changes during the execution of the function and only the final write to `d_o` is relevant. The other writes are overwritten by the time the function completes.

Vitis HLS matches the behavior of the `gcc` compiler and optimizes these reads and writes into a single read operation and a single write operation. When the RTL is examined, there is only a single read and write operation on each port.

The fundamental issue with this design is that the test bench and design do not adequately model how you expect the RTL ports to be implemented:

- You expect RTL ports that read and write multiple times during a transaction (and can stream the data in and out).

- The test bench supplies only a single input value and returns only a single output value. A C/C++ simulation of [Multi-Access Pointers on the Interface](#) shows the following results, which demonstrates that each input is being accumulated four times. The same value is being read once and accumulated each time. It is not four separate reads.

Din	Dout
0	0
1	4
2	8
3	12

To make this design read and write to the RTL ports multiple times, use a `volatile` qualifier as shown in [Multi-Access Pointers on the Interface](#). The `volatile` qualifier tells the C/C++ compiler and Vitis HLS to make no assumptions about the pointer accesses, and to not optimize them away. That is, the data is volatile and might change.

The `volatile` qualifier:

- Prevents pointer access optimizations.
- Results in an RTL design that performs the expected four reads on input port `d_i` and two writes to output port `d_o`.

Even if the `volatile` keyword is used, the coding style of accessing a pointer multiple times still has an issue in that the function and test bench do not adequately model multiple distinct reads and writes. In this case, four reads are performed, but the same data is read four times. There are two separate writes, each with the correct data, but the test bench captures data only for the final write.



TIP: In order to see the intermediate accesses, use `cosim_design -trace_level` to create a trace file during RTL simulation and view the trace file in the appropriate viewer.

The Multi-Access volatile pointer interface can be implemented with wire interfaces. If a FIFO interface is specified, Vitis HLS creates an RTL test bench to stream new data on each read. Because no new data is available from the test bench, the RTL fails to verify. The test bench does not correctly model the reads and writes.

Modeling Streaming Data Interfaces

Unlike software, the concurrent nature of hardware systems allows them to take advantage of streaming data. Data is continuously supplied to the design and the design continuously outputs data. An RTL design can accept new data before the design has finished processing the existing data.

As [Understanding Volatile Data](#) shows, modeling streaming data in software is non-trivial, especially when writing software to model an existing hardware implementation (where the concurrent/streaming nature already exists and needs to be modeled).

There are several possible approaches:

- Add the `volatile` qualifier as shown in the Multi-Access Volatile Pointer Interface example. The test bench does not model unique reads and writes, and RTL simulation using the original C/C++ test bench might fail, but viewing the trace file waveforms shows that the correct reads and writes are being performed.
- Modify the code to model explicit unique reads and writes. See the following example.
- Modify the code to using a streaming data type. A streaming data type allows hardware using streaming data to be accurately modeled.

The following code example has been updated to ensure that it reads four unique values from the test bench and write two unique values. Because the pointer accesses are sequential and start at location zero, a streaming interface type can be used during synthesis.

```
#include "pointer_stream_good.h"

void pointer_stream_good ( volatile dout_t *d_o, volatile din_t *d_i) {
    din_t acc = 0;

    acc += *d_i;
    acc += *(d_i+1);
    *d_o = acc;
    acc += *(d_i+2);
    acc += *(d_i+3);
    *(d_o+1) = acc;
}
```

The test bench is updated to model the fact that the function reads four unique values in each transaction. This new test bench models only a single transaction. To model multiple transactions, the input data set must be increased and the function called multiple times.

```
#include "pointer_stream_good.h"

int main () {
    din_t d_i[4];
    dout_t d_o[4];
    int i, retval=0;
    FILE *fp;

    // Create input data
    for (i=0;i<4;i++) {
        d_i[i] = i;
    }

    // Call the function to operate on the data
    pointer_stream_good(d_o,d_i);

    // Save the results to a file
    fp=fopen(result.dat,w);
    for (i=0;i<4;i++) {
        if (i<2)
            fprintf(fp, %d %d\n, d_i[i], d_o[i]);
        else
            fprintf(fp, %d \n, d_i[i]);
    }
    fclose(fp);

    // Compare the results file with the golden results
}
```

```

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

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

```

The test bench validates the algorithm with the following results, showing that:

- There are two outputs from a single transaction.
- The outputs are an accumulation of the first two input reads, plus an accumulation of the next two input reads and the previous accumulation.

Din	Dout
0	1
1	6
2	
3	

- The final issue to be aware of when pointers are accessed multiple time at the function interface is RTL simulation modeling.

Multi-Access Pointers and RTL Simulation

When pointers on the interface are accessed multiple times, to read or write, Vitis HLS cannot determine from the function interface how many reads or writes are performed. Neither of the arguments in the function interface informs Vitis HLS how many values are read or written.

```
void pointer_stream_good (volatile dout_t *d_o, volatile din_t *d_i)
```

Unless the code informs Vitis HLS how many values are required (for example, the maximum size of an array), the tool assumes a single value and models C/RTL co-simulation for only a single input and a single output. If the RTL ports are actually reading or writing multiple values, the RTL co-simulation stalls. RTL co-simulation models the external producer and consumer blocks that are connected to the RTL design through the port interface. If it requires more than a single value, the RTL design stalls when trying to read or write more than one value because there is currently no value to read, or no space to write.

When multi-access pointers are used at the interface, Vitis HLS must be informed of the required number of reads or writes on the interface. Manually specify the INTERFACE pragma or directive for the pointer interface, and set the `depth` option to the required depth.

For example, argument `d_i` in the code sample above requires a FIFO depth of four. This ensures RTL co-simulation provides enough values to correctly verify the RTL.

Vector Data Types

HLS Vector Type for SIMD Operations

The Vitis™ HLS library provides the reference implementation for the `hls::vector<T, N>` type which represent a single-instruction multiple-data (SIMD) vector of `N` elements of type `T`:

- `T` can be a user-defined type which must provide common arithmetic operations.
- `N` must be a positive integer.
- The best performance is achieved when both the bit-width of `T` and `N` are integer powers of 2.

The vector data type is provided to easily model and synthesize SIMD-type vector operations. Many operators are overloaded to provide SIMD behavior for vector types. SIMD vector operations are characterized by two parameters:

1. The type of elements that the vector holds.
2. The number of elements that the vector holds.

The following example defines how the GCC compiler extensions enable support for vector type operations. It essentially provides a method to define the element type through `typedef`, and uses an attribute to specify the vector size. This new `typedef` can be used to perform operations on the vector type which are compiled for execution on software targets supporting SIMD instructions. Generally, the size of the vector used during `typedef` is specific to targets.

```
typedef int t_simd __attribute__((vector_size(16)));
t_simd a, b, c;
c = a + b;
```

In the case of Vitis HLS vector data type, SIMD operations can be modeled on similar lines. Vitis HLS provides a template type `hls::vector` that can be used to define SIMD operands. All the operation performed using this type are mapped to hardware during synthesis that will execute these operations in parallel. These operations can be carried out in a loop which can be pipelined with `II=1`. The following example shows how an eight element vector of integers is defined and used:

```
typedef hls::vector<int, 8> t_int8Vec;
t_int8Vec intVectorA, intVectorB;
.

.

void processVecStream(hls::stream<t_int8Vec>
&inVecStream1, hls::stream<t_int8Vec> &inVecStream2, hls::stream<int8Vec>
&outVecStream)
{
    for(int i=0;i<32;i++)
    {
        #pragma HLS pipeline II=1
        t_int8Vec aVec = inVecStream1.read();
        t_int8Vec bVec = inVecStream2.read();
```

```
//performs a vector operation on 8 integers in parallel
t_int8Vec cVec = aVec * bVec;
outVecStream.write(cVec);
}
```

Vector Data Type Usage

Vitis HLS vector data type can be defined as follows, where `T` is a primitive or user-defined type with most of the arithmetic operations defined on it. `N` is an integer greater than zero. Once a vector type variable is declared it can be used like any other primitive type variable to perform arithmetic and logic operations.

```
#include <hls_vector.h>
hls::vector<T,N> aVec;
```

Memory Layout

For any Vitis HLS vector type defined as `hls::vector<T,N>`, the storage is guaranteed to be contiguous of size `sizeof(T) * N` and aligned to the greatest power of 2 such that the allocated size is at least `sizeof(T) * N`. In particular, when `N` is a power of 2 and `sizeof(T)` is a power of 2, `vector<T, N>` is aligned to its total size. This matches vector implementation on most architectures.



TIP: When `sizeof(T) * N` is an integer power of 2, the allocated size will be exactly `sizeof(T) * N`, otherwise the allocated size will be larger to make alignment possible.

The following example shows the definition of a vector class that aligns itself as described above.

```
constexpr size_t gp2(size_t N)
{
    return (N > 0 && N % 2 == 0) ? 2 * gp2(N / 2) : 1;

template<typename T, size_t N> class alignas(gp2(sizeof(T) * N)) vector
{
    std::array<T, N> data;
};
```

Following are different examples of alignment:

```
hls::vector<char,8> char8Vec; // aligns on 8 Bytes boundary
hls::vector<int,8> int8Vec; // aligns on 32 byte boundary
```

Requirements and Dependencies

Vitis HLS vector types requires support for C++ 14 or later. It has the following dependencies on the standard headers:

- <array>
 - `std::array<T, N>`

- <cassert>
 - assert
- <initializer_list>
 - std::initializer_list<T>

Supported Operations

- Initialization:

```

hls::vector<int, 4> x; // uninitialized
hls::vector<int, 4> y = 10; // scalar initialized: all elements set to 10
hls::vector<int, 4> z = {0, 1, 2, 3}; // initializer list (must have 4
elements)
hls::vector<ap_int, 4> a; // uninitialized arbitrary precision data type

```

- Access:

The operator[] enables access to individual elements of the vector, similar to a standard array:

```

x[i] = ...; // set the element at index i
... = x[i]; // value of the element at index i

```

- Arithmetic:

They are defined recursively, relying on the matching operation on T.

Table 16: Arithmetic Operation

Operation	In Place	Expression	Reduction (Left Fold)
Addition	<code>+ =</code>	<code>+</code>	<code>reduce_add</code>
Subtraction	<code>- =</code>	<code>-</code>	<i>non-associative</i>
Multiplication	<code>* =</code>	<code>*</code>	<code>reduce_mult</code>
Division	<code>/ =</code>	<code>/</code>	<i>non-associative</i>
Remainder	<code>% =</code>	<code>%</code>	<i>non-associative</i>
Bitwise AND	<code>& =</code>	<code>&</code>	<code>reduce_and</code>
Bitwise OR	<code> =</code>	<code> </code>	<code>reduce_or</code>
Bitwise XOR	<code>^ =</code>	<code>^</code>	<code>reduce_xor</code>
Shift Left	<code><< =</code>	<code><<</code>	<i>non-associative</i>
Shift Right	<code>>> =</code>	<code>>></code>	<i>non-associative</i>
Pre-increment	<code>++ x</code>	<i>none</i>	<i>unary operator</i>
Pre-decrement	<code>-- x</code>	<i>none</i>	<i>unary operator</i>
Post-increment	<code>x ++</code>	<i>none</i>	<i>unary operator</i>
Post-decrement	<code>x --</code>	<i>none</i>	<i>unary operator</i>

- Comparison:

Lexicographic order on vectors (returns bool):

Table 17: Operation

Operation	Expression
Less than	<
Less or equal	<=
Equal	==
Different	!=
Greater or equal	>=
Greater than	>

C++ Classes and Templates

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

```
#include "cpp_FIR.h"

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

    cout << fir1;

    return fir1(x);
}
```



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

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

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

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

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

#define N 85

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

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

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

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

    loop: for (i = N-1; i >= 0; i--) {
        if (i == 0) {
            m = x;
            shift_reg[0] = x;
        } else {
            m = shift_reg[i-1];
            if (i != (N-1))
                shift_reg[i] = shift_reg[i - 1];
        }
        acc += m * c[i];
    }
    return acc;
}

// Operator for displaying results
template<class coef_T, class data_T, class acc_T>
ostream& operator<<(ostream& o, const CFir<coef_T, data_T, acc_T> &f) {
```

```

for (int i = 0; i < (sizeof(f.shift_reg)/sizeof(data_T)); i++) {
    o << shift_reg[<< i << ]= << f.shift_reg[i] << endl;
}
o << _____ << endl;
return o;
}

data_t cpp_FIR(data_t x);

```

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

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

```

#include "cpp_FIR.h"

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

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

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

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

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

    // Return 0 if the test
    return retval;
}

```

C++ Test Bench for `cpp_FIR`

To apply directives to objects defined in a class:

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

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

Global Variables and Classes

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

```

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

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

// Use k on line 73 static int k;

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

        if (col==0) {
            SHIFT:for (k = N-1; k >= 0; --k) {
                if (k > 0)
                    shift[k] = shift[k-1];
                else
                    shift[k] = data;
            }
            *dataOut = shift_output;
            shift_output = shift[N-1];
        }
        *pcout = (shift[4*col]* coeff) + pcin;
    }
};

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

```

```

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

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

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

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

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

}
sample_count++;
}
    
```

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

```

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

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

Templates

Vitis HLS supports the use of templates in C++ for synthesis. Vitis HLS does not support templates for the top-level function.



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

Using Templates to Create Unique Instances

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

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

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

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

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

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

Using Templates for Recursion

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

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

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

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

```

        return b;
    }

};

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

```

Assertions

 **IMPORTANT!** Usage of assertions can cause bad logic to be created as assertions can have side effects that are not obvious to the user. They can also affect compiler optimizations from not happening depending on the complexity of code inside the assert statement.

The assert macro in C/C++ is supported for synthesis when used to assert range information. For example, the upper limit of variables and loop-bounds.

When variable loop bounds are present, Vitis HLS cannot determine the latency for all iterations of the loop and reports the latency with a question mark. The `tripcount` directive can inform Vitis HLS of the loop bounds, but this information is only used for reporting purposes and does not impact the result of synthesis (the same sized hardware is created, with or without the `tripcount` directive).

The following code example shows how assertions can inform Vitis HLS about the maximum range of variables, and how those assertions are used to produce more optimal hardware.

Before using assertions, the header file that defines the `assert` macro must be included. In this example, this is included in the header file.

```

#ifndef _loop_sequential_assert_H_
#define _loop_sequential_assert_H_

#include <stdio.h>
#include <assert.h>
#include ap_int.h
#define N 32

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

void loop_sequential_assert(din_t A[N], din_t B[N], dout_t X[N], dout_t
Y[N], dsel_t xlimit, dsel_t ylimit);

#endif

```

In the main code two `assert` statements are placed before each of the loops.

```
assert(xlimit<32);
...
assert(ylimit<16);
...
```

These assertions:

- Guarantee that if the assertion is false and the value is greater than that stated, the C/C++ simulation will fail. This also highlights why it is important to simulate the C/C++ code before synthesis: confirm the design is valid before synthesis.
- Inform Vitis HLS that the range of this variable will not exceed this value and this fact can optimize the variables size in the RTL and in this case, the loop iteration count.

The following code example shows these assertions.

```
#include "loop_sequential_assert.h"

void loop_sequential_assert(din_t A[N], din_t B[N], dout_t X[N], dout_t
Y[N], dsel_t
xlimit, dsel_t ylimit) {

    dout_t X_accum=0;
    dout_t Y_accum=0;
    int i,j;

    assert(xlimit<32);
    SUM_X:for (i=0;i<=xlimit; i++) {
        X_accum += A[i];
        X[i] = X_accum;
    }

    assert(ylimit<16);
    SUM_Y:for (i=0;i<=ylimit; i++) {
        Y_accum += B[i];
        Y[i] = Y_accum;
    }
}
```

Except for the `assert` macros, this code is the same as that shown in [Loop Parallelism](#). There are two important differences in the synthesis report after synthesis.

Without the `assert` macros, the report is as follows, showing that the loop Trip Count can vary from 1 to 256 because the variables for the loop-bounds are of data type `d_sel` that is an 8-bit variable.

```
* Loop Latency:
+-----+-----+-----+
|Target |II|Trip Count|Pipelined|
+-----+-----+-----+
|- SUM_X|1 ~ 256|no      |
|- SUM_Y|1 ~ 256|no      |
+-----+-----+-----+
```

In the version with the `assert` macros, the report shows the loops `SUM_X` and `SUM_Y` reported Trip Count of 32 and 16. Because the assertions assert that the values will never be greater than 32 and 16, Vitis HLS can use this in the reporting.

* Loop Latency:			
Target	II	Trip Count	Pipelined
- SUM_X	1	~ 32	no
- SUM_Y	1	~ 16	no

In addition, and unlike using the `tripcount` directive, the `assert` statements can provide more optimal hardware. In the case without assertions, the final hardware uses variables and counters that are sized for a maximum of 256 loop iterations.

* Expression:				
Operation	Variable Name	DSP48E	IFF	LUT
+	X_accum_1_fu_182_p2	0	0	13
+	Y_accum_1_fu_209_p2	0	0	13
+	indvar_next6_fu_158_p2	0	0	9
+	indvar_next_fu_194_p2	0	0	9
+	tmp1_fu_172_p2	0	0	9
+	tmp_fu_147_p2	0	0	9
icmp	exitcond1_fu_189_p2	0	0	9
icmp	exitcond_fu_153_p2	0	0	9
Total		0	0	80

The code which asserts the variable ranges are smaller than the maximum possible range results in a smaller RTL design.

* Expression:				
Operation	Variable Name	DSP48E	IFF	LUT
+	X_accum_1_fu_176_p2	0	0	13
+	Y_accum_1_fu_207_p2	0	0	13
+	i_2_fu_158_p2	0	0	6
+	i_3_fu_192_p2	0	0	5
icmp	tmp_2_fu_153_p2	0	0	7
icmp	tmp_9_fu_187_p2	0	0	6
Total		0	0	50

Assertions can indicate the range of any variable in the design. It is important to execute a C/C++ simulation that covers all possible cases when using assertions. This will confirm that the assertions that Vitis HLS uses are valid.

Examples of Hardware Efficient C++ Code

When C++ code is compiled for a CPU, the compiler transforms and optimizes the C++ code into a set of CPU machine instructions. In many cases, the developers work is done at this stage. If however, there is a need for performance the developer will seek to perform some or all of the following:

- Understand if any additional optimizations can be performed by the compiler.
- Seek to better understand the processor architecture and modify the code to take advantage of any architecture specific behaviors (for example, reducing conditional branching to improve instruction pipelining).
- Modify the C++ code to use CPU-specific intrinsics to perform key operations in parallel (for example, Arm® NEON intrinsics).

The same methodology applies to code written for a DSP or a GPU, and when using an FPGA: an FPGA is simply another target.

C++ code synthesized by Vitis HLS will execute on an FPGA and provide the same functionality as the C++ simulation. In some cases, the developers work is done at this stage.

Typically however, an FPGA is selected to implement the C++ code due to the superior performance of the FPGA - the massively parallel architecture of an FPGA allows it to perform operations much faster than the inherently sequential operations of a processor - and users typically wish to take advantage of that performance.

The focus here is on understanding the impact of the C++ code on the results which can be achieved and how modifications to the C++ code can be used to extract the maximum advantage from the first three items in this list.

Typical C++ Code for a Convolution Function

A standard convolution function applied to an image is used here to demonstrate how the C++ code can negatively impact the performance which is possible from an FPGA. In this example, a horizontal and then vertical convolution is performed on the data. Since the data at edge of the image lies outside the convolution windows, the final step is to address the data around the border.

The algorithm structure can be summarized as follows:

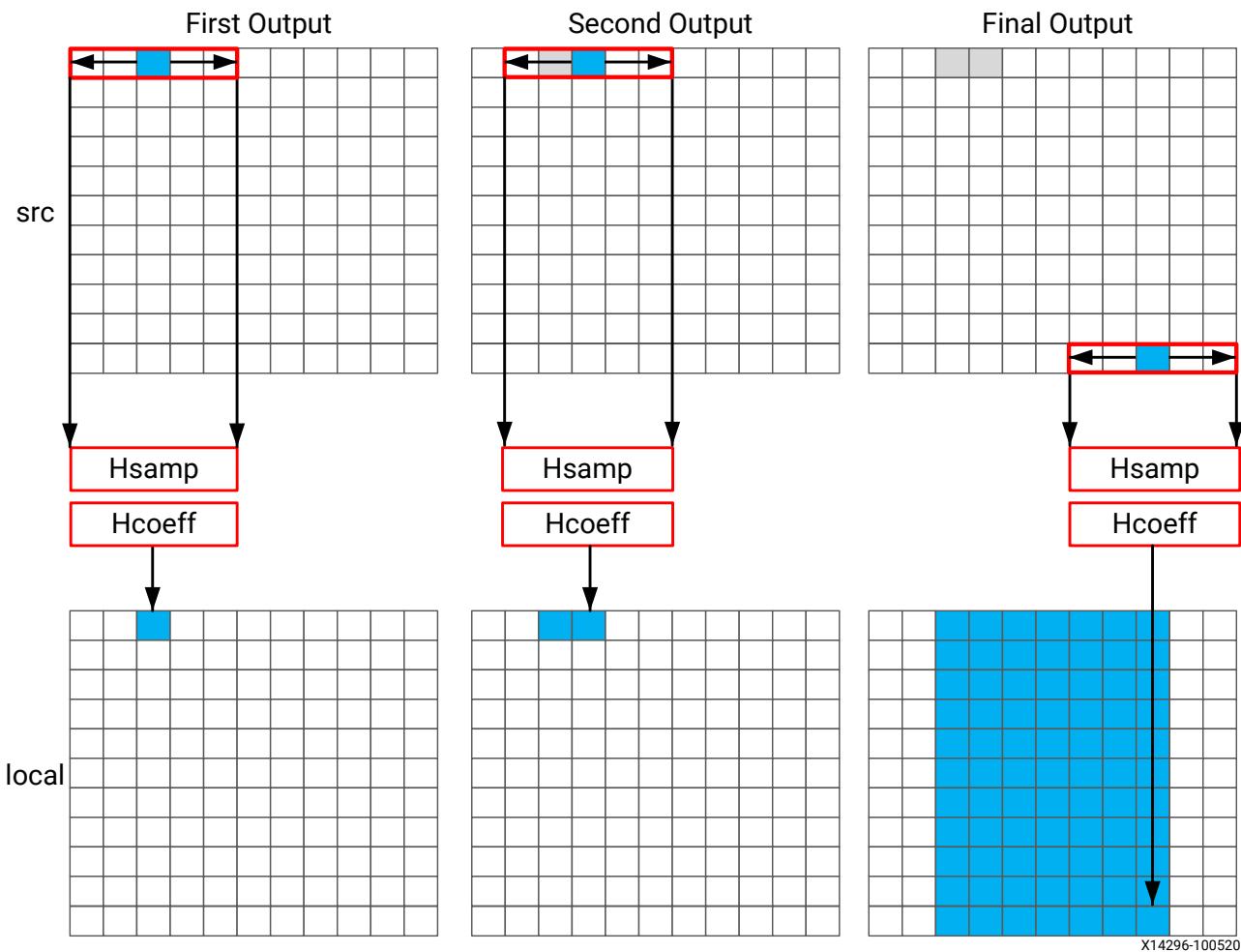
```
template<typename T, int K>
static void convolution_orig(
    int width,
    int height,
    const T *src,
    T *dst,
    const T *hcoeff,
```

```
const T *vcoeff) {  
  
T local[MAX_IMG_ROWS*MAX_IMG_COLS];  
  
// Horizontal convolution  
HconvH:for(int col = 0; col < height; col++){  
    HconvW:for(int row = border_width; row < width - border_width; row++){  
        Hconv:for(int i = - border_width; i <= border_width; i++){  
            }  
        }  
    }  
// Vertical convolution  
VconvH:for(int col = border_width; col < height - border_width; col++){  
    VconvW:for(int row = 0; row < width; row++){  
        Vconv:for(int i = - border_width; i <= border_width; i++){  
            }  
        }  
    }  
// Border pixels  
Top_Border:for(int col = 0; col < border_width; col++){  
}  
Side_Border:for(int col = border_width; col < height - border_width; col++){  
}  
Bottom_Border:for(int col = height - border_width; col < height; col++){  
}  
}
```

Horizontal Convolution

The first step in this is to perform the convolution in the horizontal direction as shown in the following figure.

Figure 54: Horizontal Convolution



The convolution is performed using K samples of data and K convolution coefficients. In the figure above, K is shown as 5 however the value of K is defined in the code. To perform the convolution, a minimum of K data samples are required. The convolution window cannot start at the first pixel, because the window would need to include pixels which are outside the image.

By performing a symmetric convolution, the first K data samples from input `src` can be convolved with the horizontal coefficients and the first output calculated. To calculate the second output, the next set of K data samples are used. This calculation proceeds along each row until the final output is written.

The final result is a smaller image, shown above in blue. The pixels along the vertical border are addressed later.

The C/C++ code for performing this operation is shown below.

```
const int conv_size = K;
const int border_width = int(conv_size / 2);

#ifndef __SYNTHESIS__
    T * const local = new T[MAX_IMG_ROWS*MAX_IMG_COLS];
#else // Static storage allocation for HLS, dynamic otherwise
    T local[MAX_IMG_ROWS*MAX_IMG_COLS];
#endif

Clear_Local:for(int i = 0; i < height * width; i++){
    local[i]=0;
}
// Horizontal convolution
HconvH:for(int col = 0; col < height; col++){
    HconvWfor(int row = border_width; row < width - border_width; row++) {
        int pixel = col * width + row;
        Hconv:for(int i = - border_width; i <= border_width; i++) {
            local[pixel] += src[pixel + i] * hcoeff[i + border_width];
        }
    }
}
```

Note: Only use the `__SYNTHESIS__` macro in the code to be synthesized. Do not use this macro in the test bench, because it is not obeyed by C/C++ simulation or C/C++ RTL co-simulation.

The code is straight forward and intuitive. There are already however some issues with this C/C++ code and three which will negatively impact the quality of the hardware results.

The first issue is the requirement for two separate storage requirements. The results are stored in an internal `local` array. This requires an array of `HEIGHT*WIDTH` which for a standard video image of `1920*1080` will hold 2,073,600 values. On some Windows systems, it is not uncommon for this amount of local storage to create issues. The data for a `local` array is placed on the stack and not the heap which is managed by the OS.

A useful way to avoid such issues is to use the `__SYNTHESIS__` macro. This macro is automatically defined when synthesis is executed. The code shown above will use the dynamic memory allocation during C/C++ simulation to avoid any compilation issues and only use the static storage during synthesis. A downside of using this macro is the code verified by C/C++ simulation is not the same code which is synthesized. In this case however, the code is not complex and the behavior will be the same.

The first issue for the quality of the FPGA implementation is the array `local`. Because this is an array it will be implemented using internal FPGA block RAM. This is a very large memory to implement inside the FPGA. It may require a larger and more costly FPGA device. The use of block RAM can be minimized by using the DATAFLOW optimization and streaming the data through small efficient FIFOs, but this will require the data to be used in a streaming manner.

The next issue is the initialization for array `local`. The loop `Clear_Local` is used to set the values in array `local` to zero. Even if this loop is pipelined, this operation will require approximately 2 million clock cycles (`HEIGHT*WIDTH`) to implement. This same initialization of the data could be performed using a temporary variable inside loop `HConv` to initialize the accumulation before the write.

Finally, the throughput of the data is limited by the data access pattern.

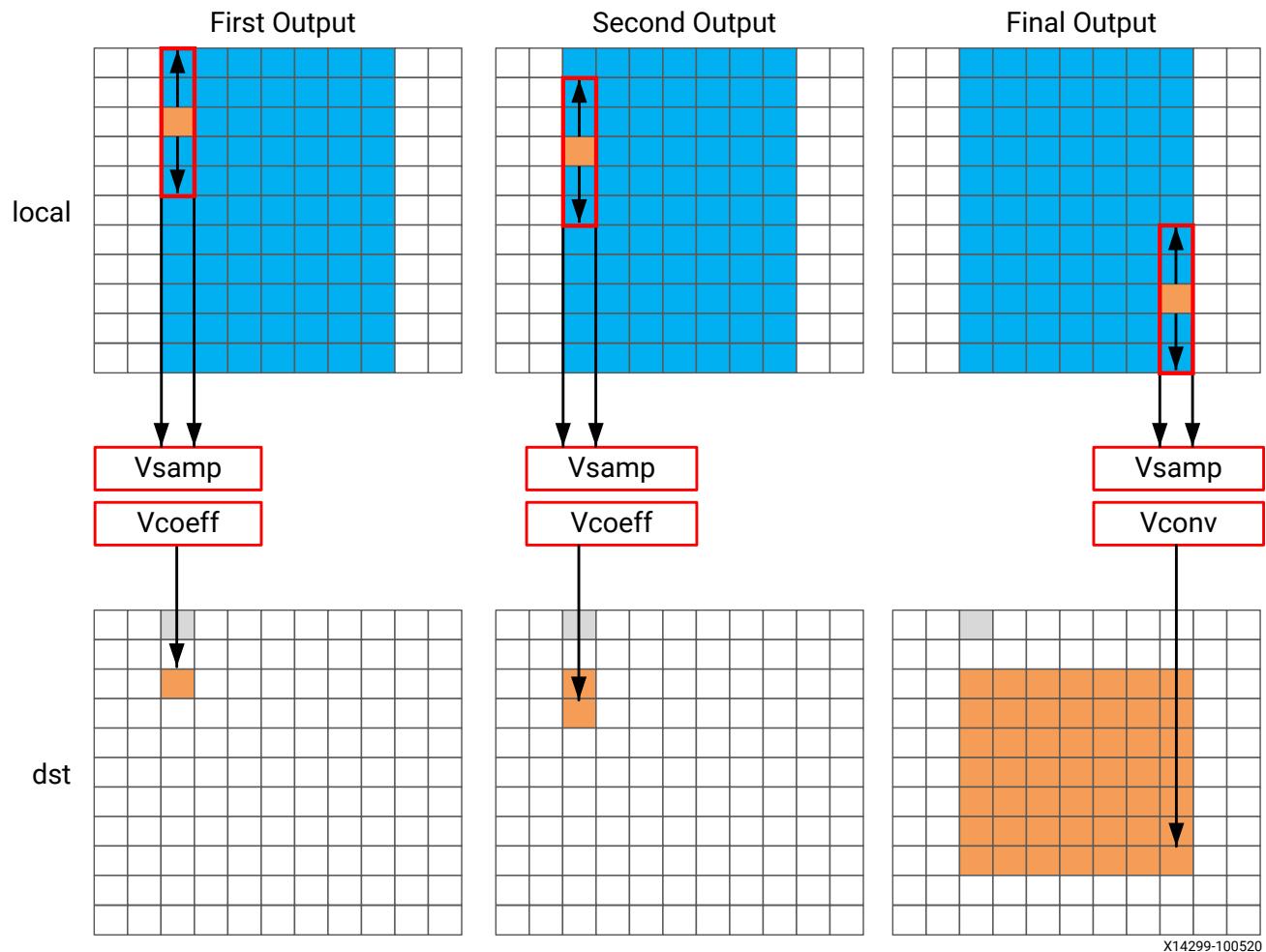
- For the first output, the first K values are read from the input.
- To calculate the second output, the same K-1 values are re-read through the data input port.
- This process of re-reading the data is repeated for the entire image.

One of the keys to a high-performance FPGA is to minimize the access to and from the top-level function arguments. The top-level function arguments become the data ports on the RTL block. With the code shown above, the data cannot be streamed directly from a processor using a DMA operation, because the data is required to be re-read time and again. Re-reading inputs also limits the rate at which the FPGA can process samples.

Vertical Convolution

The next step is to perform the vertical convolution shown in the following figure.

Figure 55: Vertical Convolution



The process for the vertical convolution is similar to the horizontal convolution. A set of K data samples is required to convolve with the convolution coefficients, V_{coeff} in this case. After the first output is created using the first K samples in the vertical direction, the next set K values are used to create the second output. The process continues down through each column until the final output is created.

After the vertical convolution, the image is now smaller than the source image src due to both the horizontal and vertical border effect.

The code for performing these operations is:

```

Clear_Dst:for(int i = 0; i < height * width; i++){
    dst[i]=0;
}
// Vertical convolution
VconvH:for(int col = border_width; col < height - border_width; col++){
    VconvW:for(int row = 0; row < width; row++){
        int pixel = col * width + row;
    }
}

```

```
Vconv:for(int i = - border_width; i <= border_width; i++) {
    int offset = i * width;
    dst[pixel] += local[pixel + offset] * vcoeff[i + border_width];
}
```

This code highlights similar issues to those already discussed with the horizontal convolution code.

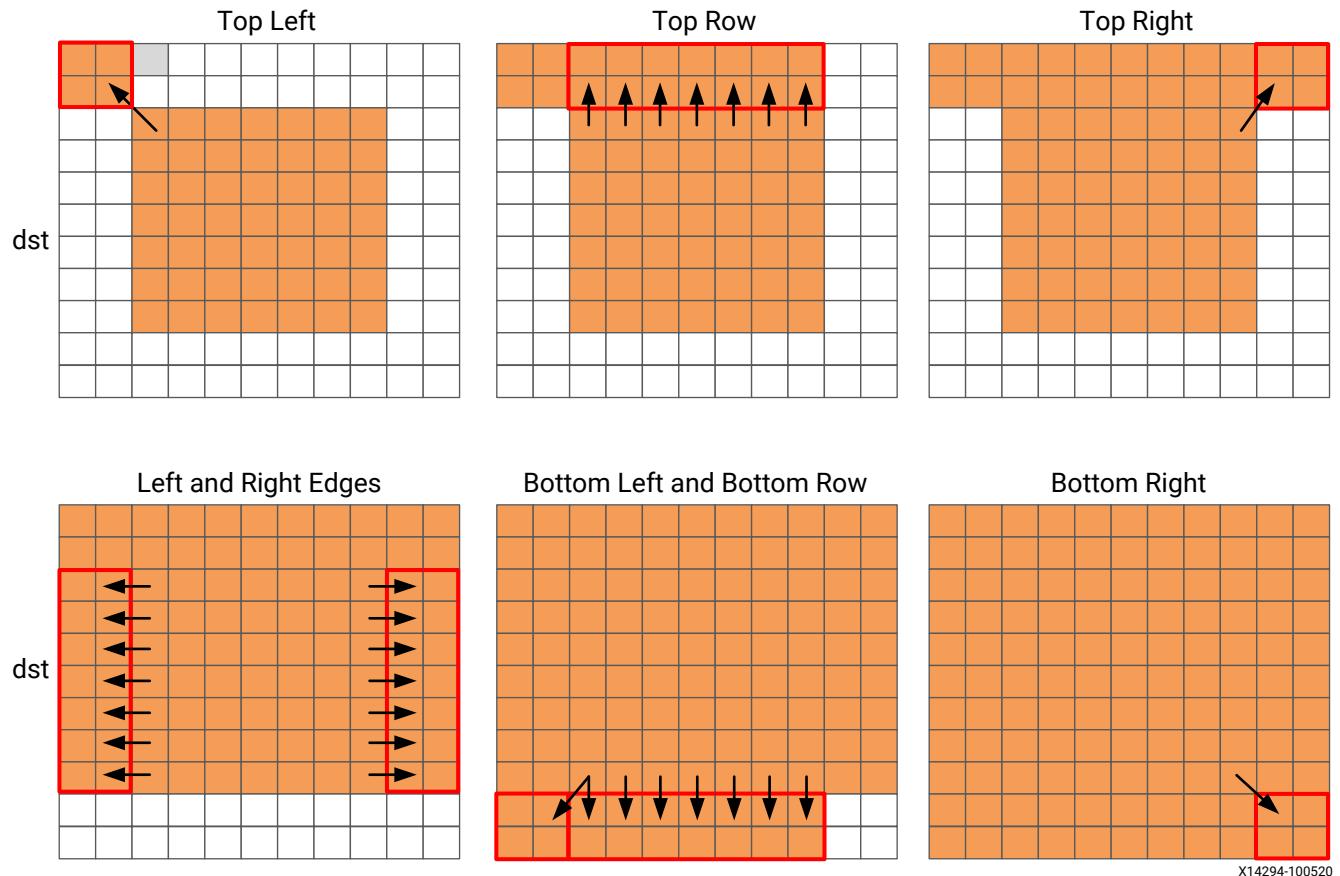
- Many clock cycles are spent to set the values in the output image `dst` to zero. In this case, approximately another 2 million cycles for a 1920*1080 image size.
- There are multiple accesses per pixel to re-read data stored in array `local`.
- There are multiple writes per pixel to the output array/port `dst`.

Another issue with the code above is the access pattern into array `local`. The algorithm requires the data on row K to be available to perform the first calculation. Processing data down the rows before proceeding to the next column requires the entire image to be stored locally. In addition, because the data is not streamed out of array `local`, a FIFO cannot be used to implement the memory channels created by DATAFLOW optimization. If DATAFLOW optimization is used on this design, this memory channel requires a ping-pong buffer: this doubles the memory requirements for the implementation to approximately 4 million data samples all stored locally on the FPGA.

Border Pixels

The final step in performing the convolution is to create the data around the border. These pixels can be created by simply re-using the nearest pixel in the convolved output. The following figures shows how this is achieved.

Figure 56: Convolution Border Samples



The border region is populated with the nearest valid value. The following code performs the operations shown in the figure.

```

int border_width_offset = border_width * width;
int border_height_offset = (height - border_width - 1) * width;
// Border pixels
Top_Border:for(int col = 0; col < border_width; col++){
    int offset = col * width;
    for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + border_width];
    }
    for(int row = border_width; row < width - border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + row];
    }
    for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + width - border_width - 1];
    }
}

Side_Border:for(int col = border_width; col < height - border_width; col++){
    int offset = col * width;
    for(int row = 0; row < border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + border_width];
    }
    for(int row = border_width; row < width - border_width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + row];
    }
    for(int row = width - border_width; row < width; row++){
        int pixel = offset + row;
        dst[pixel] = dst[border_width_offset + width - border_width - 1];
    }
}

```

```

for(int row = 0; row < border_width; row++) {
    int pixel = offset + row;
    dst[pixel] = dst[offset + border_width];
}
for(int row = width - border_width; row < width; row++) {
    int pixel = offset + row;
    dst[pixel] = dst[offset + width - border_width - 1];
}
}

Bottom_Border:for(int col = height - border_width; col < height; col++) {
    int offset = col * width;
    for(int row = 0; row < border_width; row++) {
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + border_width];
    }
    for(int row = border_width; row < width - border_width; row++) {
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + row];
    }
    for(int row = width - border_width; row < width; row++) {
        int pixel = offset + row;
        dst[pixel] = dst[border_height_offset + width - border_width - 1];
    }
}

```

The code suffers from the same repeated access for data. The data stored outside the FPGA in array `dst` must now be available to be read as input data re-read multiple times. Even in the first loop, `dst[border_width_offset + border_width]` is read multiple times but the values of `border_width_offset` and `border_width` do not change.

The final aspect where this coding style negatively impact the performance and quality of the FPGA implementation is the structure of how the different conditions is address. A for-loop processes the operations for each condition: top-left, top-row, etc. The optimization choice here is to:

Pipelining the top-level loops, (`Top_Border`, `Side_Border`, `Bottom_Border`) is not possible in this case because some of the sub-loops have variable bounds (based on the value of input `width`). In this case you must pipeline the sub-loops and execute each set of pipelined loops serially.

The question of whether to pipeline the top-level loop and unroll the sub-loops or pipeline the sub-loops individually is determined by the loop limits and how many resources are available on the FPGA device. If the top-level loop limit is small, unroll the loops to replicate the hardware and meet performance. If the top-level loop limit is large, pipeline the lower level loops and lose some performance by executing them sequentially in a loop (`Top_Border`, `Side_Border`, `Bottom_Border`).

As shown in this review of a standard convolution algorithm, the following coding styles negatively impact the performance and size of the FPGA implementation:

- Setting default values in arrays costs clock cycles and performance.
- Multiple accesses to read and then re-read data costs clock cycles and performance.

- Accessing data in an arbitrary or random access manner requires the data to be stored locally in arrays and costs resources.

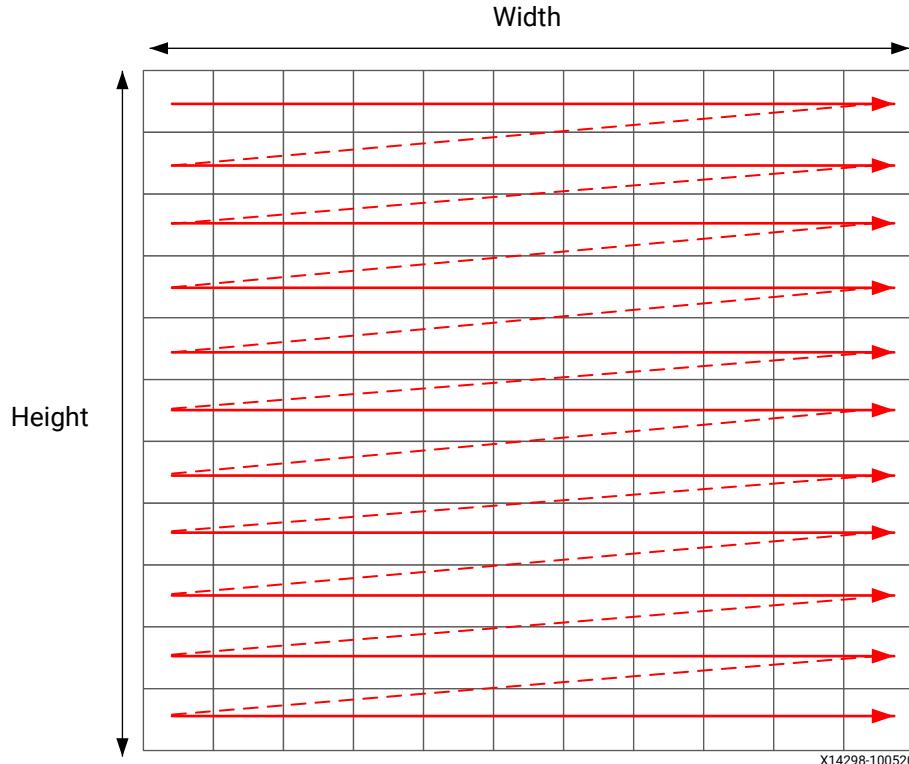
Ensuring the Continuous Flow of Data and Data Reuse

The key to implementing the convolution example reviewed in the previous section as a high-performance design with minimal resources is to consider how the FPGA implementation will be used in the overall system. The ideal behavior is to have the data samples constantly flow through the FPGA.

- Maximize the flow of data through the system. Refrain from using any coding techniques or algorithm behavior which limits the flow of data.
- Maximize the reuse of data. Use local caches to ensure there are no requirements to re-read data and the incoming data can keep flowing.

The first step is to ensure you perform optimal I/O operations into and out of the FPGA. The convolution algorithm is performed on an image. When data from an image is produced and consumed, it is transferred in a standard raster-scan manner as shown in the following figure.

Figure 57: Raster Scan Order



If the data is transferred from the CPU or system memory to the FPGA it will typically be transferred in this streaming manner. The data transferred from the FPGA back to the system should also be performed in this manner.

Using HLS Streams for Streaming Data

One of the first enhancements which can be made to the earlier code is to use the HLS stream construct, typically referred to as an `hls::stream`. An `hls::stream` object can be used to store data samples in the same manner as an array. The data in an `hls::stream` can only be accessed sequentially. In the C/C++ code, the `hls::stream` behaves like a FIFO of infinite depth.

Code written using `hls::stream` will generally create designs in an FPGA which have high-performance and use few resources because an `hls::stream` enforces a coding style which is ideal for implementation in an FPGA.

Multiple reads of the same data from an `hls::stream` are impossible. Once the data has been read from an `hls::stream` it no longer exists in the stream. This helps remove this coding practice.

If the data from an `hls::stream` is required again, it must be cached. This is another good practice when writing code to be synthesized on an FPGA.

The `hls::stream` forces the C/C++ code to be developed in a manner which is ideal for an FPGA implementation.

When an `hls::stream` is synthesized it is automatically implemented as a FIFO channel which is 1 element deep. This is the ideal hardware for connecting pipelined tasks.

There is no requirement to use `hls::stream` and the same implementation can be performed using arrays in the C/C++ code. The `hls::stream` construct does help enforce good coding practices.

With an `hls::stream` construct the outline of the new optimized code is as follows:

```
template<typename T, int K>
static void convolution_strm(
    int width,
    int height,
    hls::stream<T> &src,
    hls::stream<T> &dstd,
    const T *hcoeff,
    const T *vcoeff)
{
    hls::stream<T> hconv( "hconv" );
    hls::stream<T> vconv( "vconv" );
    // These assertions let HLS know the upper bounds of loops
    assert(height < MAX_IMG_ROWS);
    assert(width < MAX_IMG_COLS);
    assert(vconv_xlim < MAX_IMG_COLS - (K - 1));
```

```

// Horizontal convolution
HConvH:for(int col = 0; col < height; col++) {
    HConvW:for(int row = 0; row < width; row++) {
        HConv:for(int i = 0; i < K; i++) {
    }
}
}

// Vertical convolution
VConvH:for(int col = 0; col < height; col++) {
    VConvW:for(int row = 0; row < vconv_xlim; row++) {
        VConv:for(int i = 0; i < K; i++) {
    }
}

Border:for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
}
}
}

```

Some noticeable differences compared to the earlier code are:

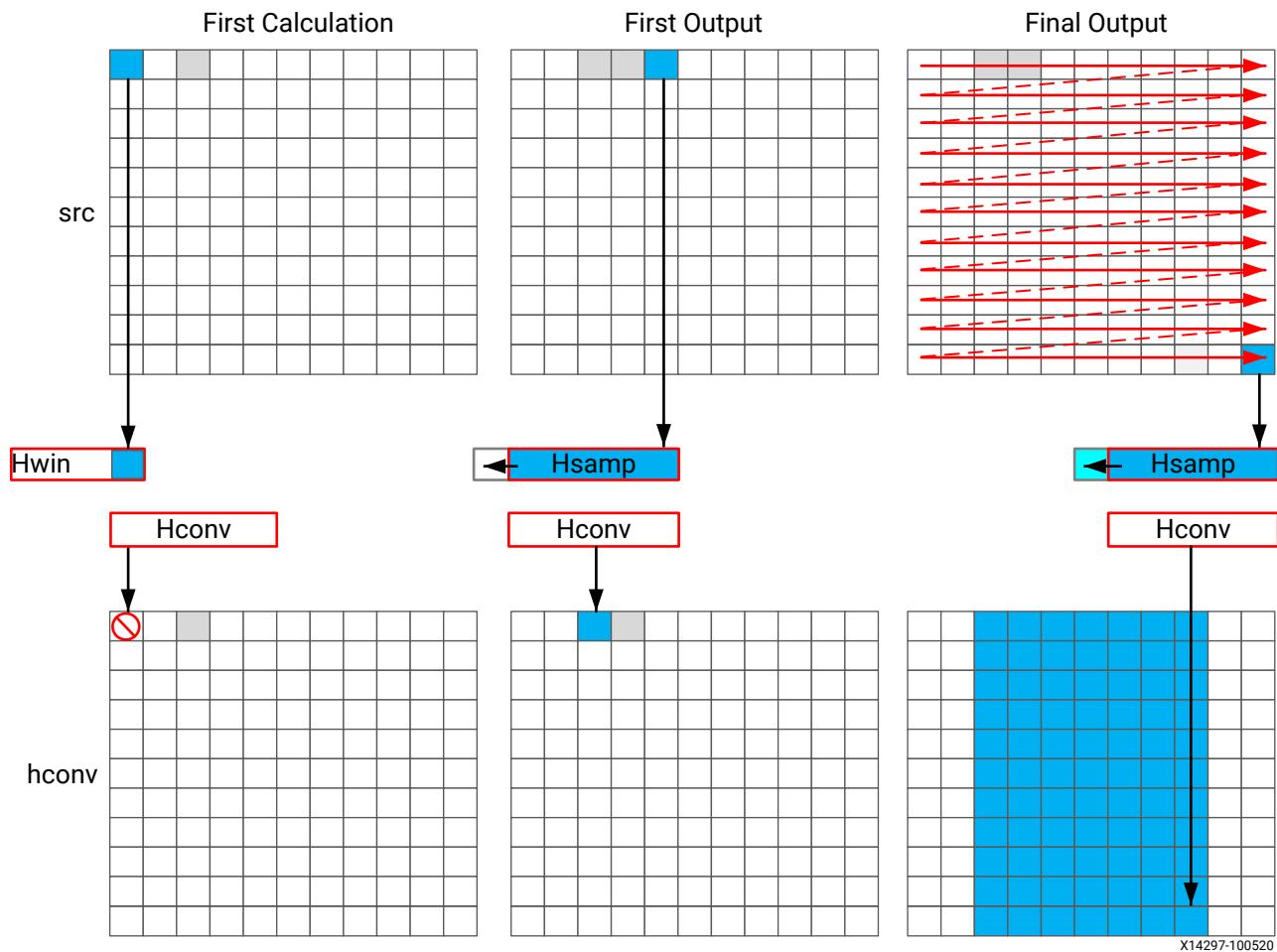
- The input and output data is now modeled as `hls::stream`.
- Instead of a single local array of size HEIGHT*WIDTH there are two internal `hls::stream` used to save the output of the horizontal and vertical convolutions.

In addition, some `assert` statements are used to specify the maximize of loop bounds. This is a good coding style which allows HLS to automatically report on the latencies of variable bounded loops and optimize the loop bounds.

Horizontal Convolution

To perform the calculation in a more efficient manner for FPGA implementation, the horizontal convolution is computed as shown in the following figure.

Figure 58: Streaming Horizontal Convolution



Using an `hls::stream` enforces the good algorithm practice of forcing you to start by reading the first sample first, as opposed to performing a random access into data. The algorithm must use the K previous samples to compute the convolution result, it therefore copies the sample into a temporary cache `hwin`. For the first calculation there are not enough values in `hwin` to compute a result, so no output values are written.

The algorithm keeps reading input samples and caching them into `hwin`. Each time it reads a new sample, it pushes an unneeded sample out of `hwin`. The first time an output value can be written is after the Kth input has been read. Now an output value can be written.

The algorithm proceeds in this manner along the rows until the final sample has been read. At that point, only the last K samples are stored in `hwin`: all that is required to compute the convolution.

The code to perform these operations is shown below.

```
// Horizontal convolution
HConvW:for(int row = 0; row < width; row++) {
    HconvW:for(int row = border_width; row < width - border_width; row++) {
        T in_val = src.read();
        T out_val = 0;
        HConv:for(int i = 0; i < K; i++) {
            hwin[i] = i < K - 1 ? hwin[i + 1] : in_val;
            out_val += hwin[i] * hcoeff[i];
        }
        if (row >= K - 1)
            hconv << out_val;
    }
}
```

An interesting point to note in the code above is use of the temporary variable `out_val` to perform the convolution calculation. This variable is set to zero before the calculation is performed, negating the need to spend 2 million clocks cycle to reset the values, as in the previous example.

Throughout the entire process, the samples in the `src` input are processed in a raster-streaming manner. Every sample is read in turn. The outputs from the task are either discarded or used, but the task keeps constantly computing. This represents a difference from code written to perform on a CPU.

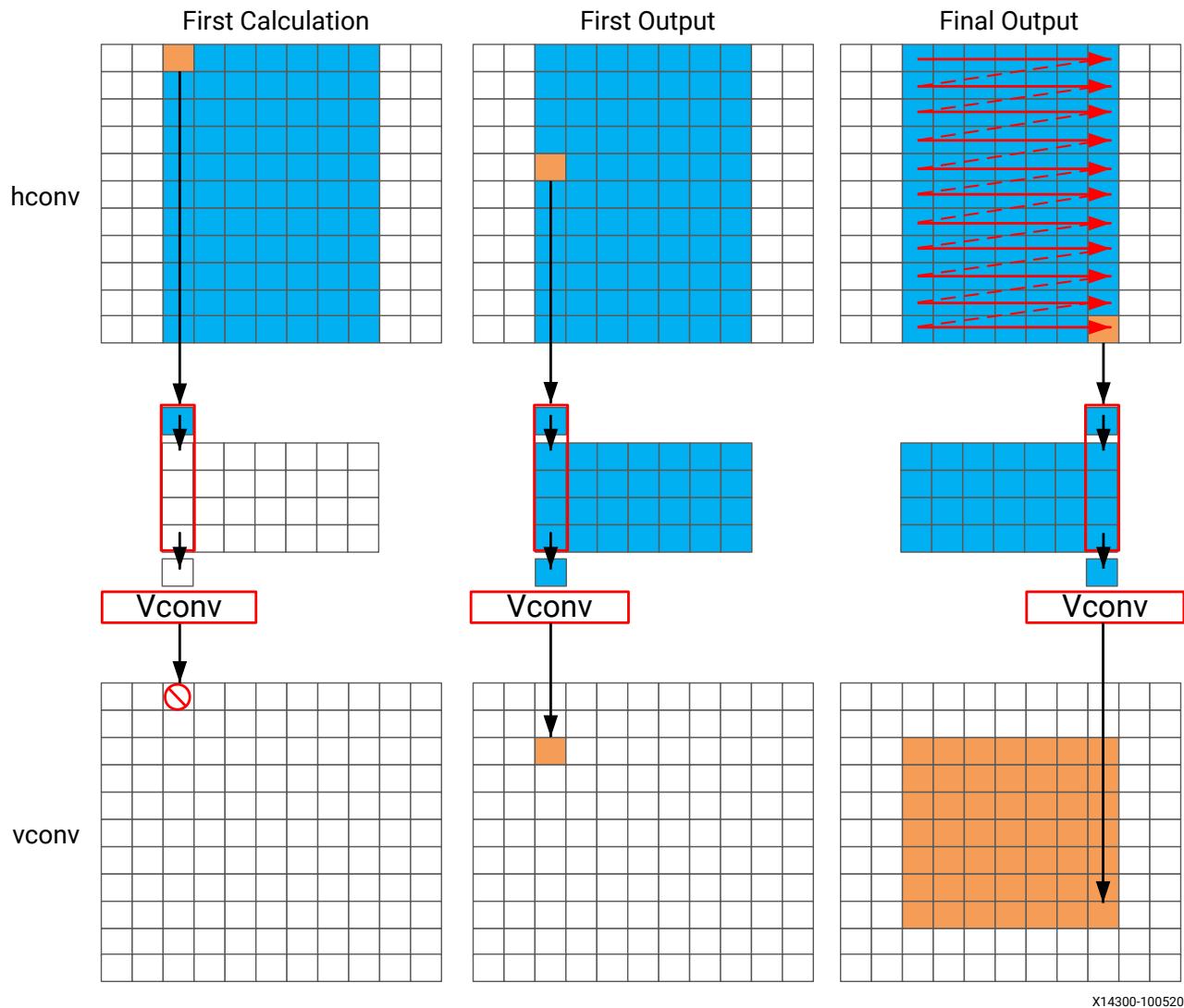
In a CPU architecture, conditional or branch operations are often avoided. When the program needs to branch it loses any instructions stored in the CPU fetch pipeline. In an FPGA architecture, a separate path already exists in the hardware for each conditional branch and there is no performance penalty associated with branching inside a pipelined task. It is simply a case of selecting which branch to use.

The outputs are stored in the `hls::stream hconv` for use by the vertical convolution loop.

Vertical Convolution

The vertical convolution represents a challenge to the streaming data model preferred by an FPGA. The data must be accessed by column but you do not wish to store the entire image. The solution is to use line buffers, as shown in the following figure.

Figure 59: Streaming Vertical Convolution



Once again, the samples are read in a streaming manner, this time from the `hls::stream` `hconv`. The algorithm requires at least $K-1$ lines of data before it can process the first sample. All the calculations performed before this are discarded.

A line buffer allows $K-1$ lines of data to be stored. Each time a new sample is read, another sample is pushed out the line buffer. An interesting point to note here is that the newest sample is used in the calculation and then the sample is stored into the line buffer and the old sample ejected out. This ensures only $K-1$ lines are required to be cached, rather than K lines. Although a line buffer does require multiple lines to be stored locally, the convolution kernel size K is always much less than the 1080 lines in a full video image.

The first calculation can be performed when the first sample on the Kth line is read. The algorithm then proceeds to output values until the final pixel is read.

```
// Vertical convolution
VConvH:for(int col = 0; col < height; col++) {
    VConvW:for(int row = 0; row < vconv_xlim; row++) {
#pragma HLS DEPENDENCE variable=linebuf type=inter dependent=false
#pragma HLS PIPELINE
        T in_val = hconv.read();
        T out_val = 0;
        VConv:for(int i = 0; i < K; i++) {
            T vwin_val = i < K - 1 ? linebuf[i][row] : in_val;
            out_val += vwin_val * vcoeff[i];
            if (i > 0)
                linebuf[i - 1][row] = vwin_val;
        }
        if (col >= K - 1)
            vconv << out_val;
    }
}
```

The code above once again process all the samples in the design in a streaming manner. The task is constantly running. The use of the `hls::stream` construct forces you to cache the data locally. This is an ideal strategy when targeting an FPGA.

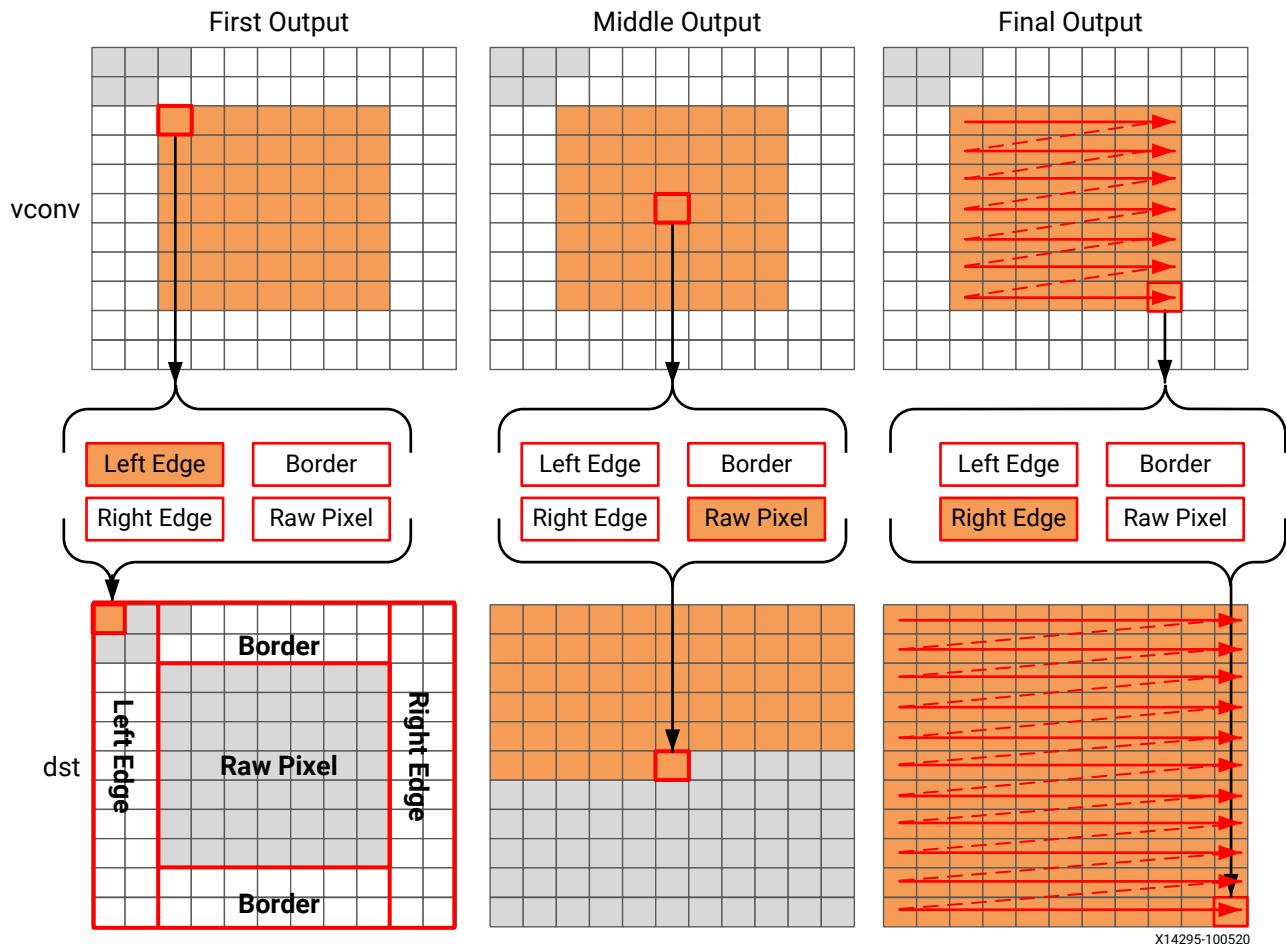
Border Pixels

The final step in the algorithm is to replicate the edge pixels into the border region. Once again, to ensure the constant flow or data and data reuse the algorithm makes use of an `hls::stream` and caching.

The following figure shows how the border samples are aligned into the image.

- Each sample is read from the `vconv` output from the vertical convolution.
- The sample is then cached as one of four possible pixel types.
- The sample is then written to the output stream.

Figure 60: Streaming Border Samples



The code for determining the location of the border pixels is:

```
Border:for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
        T pix_in, l_edge_pix, r_edge_pix, pix_out;
#pragma HLS PIPELINE
        if (i == 0 || (i > border_width && i < height - border_width)) {
            if (j < width - (K - 1)) {
                pix_in = vconv.read();
                borderbuf[j] = pix_in;
            }
            if (j == 0) {
                l_edge_pix = pix_in;
            }
            if (j == width - K) {
                r_edge_pix = pix_in;
            }
        }
        if (j <= border_width) {
            pix_out = l_edge_pix;
        } else if (j >= width - border_width - 1) {
            pix_out = r_edge_pix;
        } else {
            pix_out = pix_in;
        }
        // Process pix_out here
    }
}
```

```

    } else {
        pix_out = borderbuf[j - border_width];
    }
dst << pix_out;
}
}
}
}

```

A notable difference with this new code is the extensive use of conditionals inside the tasks. This allows the task, once it is pipelined, to continuously process data and the result of the conditionals does not impact the execution of the pipeline: the result will impact the output values but the pipeline will keep processing so long as input samples are available.

The final code for this FPGA-friendly algorithm has the following optimization directives used.

```

template<typename T, int K>
static void convolution_strm(
int width,
int height,
hls::stream<T> &src,
hls::stream<T> &dstd,
const T *hcoeff,
const T *vcoeff)
{
#pragma HLS DATAFLOW
#pragma HLS ARRAY_PARTITION variable=linebuf dim=1 type=complete

hls::stream<T> hconv("hconv");
hls::stream<T> vconv("vconv");
// These assertions let HLS know the upper bounds of loops
assert(height < MAX_IMG_ROWS);
assert(width < MAX_IMG_COLS);
assert(vconv_xlim < MAX_IMG_COLS - (K - 1));

// Horizontal convolution
HConvH:for(int col = 0; col < height; col++) {
    HConvW:for(int row = 0; row < width; row++) {
#pragma HLS PIPELINE
        HConv:for(int i = 0; i < K; i++) {
            }
        }
    }
// Vertical convolution
VConvH:for(int col = 0; col < height; col++) {
    VConvW:for(int row = 0; row < vconv_xlim; row++) {
#pragma HLS PIPELINE
#pragma HLS DEPENDENCE variable=linebuf type=inter dependent=false
        VConv:for(int i = 0; i < K; i++) {
            }
    }
}
Border:for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
#pragma HLS PIPELINE
    }
}
}

```

Each of the tasks are pipelined at the sample level. The line buffer is full partitioned into registers to ensure there are no read or write limitations due to insufficient block RAM ports. The line buffer also requires a dependence directive. All of the tasks execute in a dataflow region which will ensure the tasks run concurrently. The `hls::streams` are automatically implemented as FIFOs with 1 element.

Summary of C++ for Efficient Hardware

Minimize data input reads. Once data has been read into the block it can easily feed many parallel paths but the input ports can be bottlenecks to performance. Read data once and use a local cache if the data must be reused.

Minimize accesses to arrays, especially large arrays. Arrays are implemented in block RAM which like I/O ports only have a limited number of ports and can be bottlenecks to performance. Arrays can be partitioned into smaller arrays and even individual registers but partitioning large arrays will result in many registers being used. Use small localized caches to hold results such as accumulations and then write the final result to the array.

Seek to perform conditional branching inside pipelined tasks rather than conditionally execute tasks, even pipelined tasks. Conditionals will be implemented as separate paths in the pipeline. Allowing the data from one task to flow into with the conditional performed inside the next task will result in a higher performing system.

Minimize output writes for the same reason as input reads: ports are bottlenecks. Replicating addition ports simply pushes the issue further out into the system.

For C++ code which processes data in a streaming manner consider using `hls::streams` or `hls::stream_of_blocks`, as these will enforce good coding practices. It is much more productive to design an algorithm in C which will result in a high-performance FPGA implementation than debug why the FPGA is not operating at the performance required.

Defining Interfaces

Introduction to Interface Synthesis

The arguments of the top-level function in a Vitis HLSdesign are synthesized into interfaces and ports that group multiple signals to define the communication protocol between the HLS design and components external to the design. Vitis HLS defines interfaces automatically, using industry standards to specify the protocol used. The type of interfaces that Vitis HLS creates depends on the data type and direction of the parameters of the top-level function, the target flow for the active solution, the default interface configuration settings as specified by `config_interface`, and any specified **INTERFACE** pragmas or directives.



TIP: *Interfaces can be manually assigned using the INTERFACE pragma or directive. Refer to [Adding Pragmas and Directives](#) for more information.*

The target flows supported by Vitis HLS as described in [Vitis HLS Process Overview](#) include:

- The Vivado IP flow which is the default flow for the tool
- The Vitis Kernel flow, which is the bottom-up design flow for the Vitis Application Acceleration Development flow

You can specify the target flow when creating a project solution, as described in [Creating a New Vitis HLS Project](#), or by using the following command:

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

The interface defines three elements of the kernel:

1. The interface defines channels for data to flow into or out of the HLS design. Data can flow from a variety of sources external to the kernel or IP, such as a host application, an external camera or sensor, or from another kernel or IP implemented on the Xilinx device. The default channels for Vitis kernels are AXI adapters as described in [Interfaces for Vitis Kernel Flow](#).
2. The interface defines the port protocol that is used to control the flow of data through the data channel, defining when the data is valid and can be read or can be written, as defined in [Port-Level I/O Protocols](#).



TIP: *These port protocols can be customized in the Vivado IP flow, but are set and cannot be changed in the Vitis kernel flow, in most cases.*

3. The interface also defines the execution control scheme for the HLS design, specifying the operation of the kernel or IP as pipelined or sequential, as defined in [Block-Level Control Protocols](#).

As described in [Designing Efficient Kernels](#) the choice and configuration of interfaces is a key to the success of your design. However, Vitis HLS tries to simplify the process by selecting default interfaces for the target flows. For more information on the defaults used refer to [Interfaces for Vivado IP Flow](#) or [Interfaces for Vitis Kernel Flow](#) as appropriate to your design.

After synthesis completes you can review the mapping of the software arguments of your C/C++ code to hardware ports or interfaces in the *SW I/O Information* section of the [Synthesis Summary](#) report.

Interfaces for Vitis Kernel Flow

The Vitis kernel flow provides support for compiled kernel objects (`.xo`) for software control from a host application and by the Xilinx Run Time (XRT). As described in [Kernel Properties](#) in the *Vitis Unified Software Platform Documentation*, this flow has very specific interface requirements that Vitis HLS must meet.

Vitis HLS supports memory, stream, and register interface paradigms where each paradigm follows a certain interface protocol and uses the adapter to communicate with the external world.

- Memory Paradigm (`m_axi`): the data is accessed by the kernel through memory such as DDR, HBM, PLRAM/BRAM/URAM
- Stream Paradigm (`axis`): the data is streamed into the kernel from another streaming source, such as video processor or another kernel, and can also be streamed out of the kernel.
- Register Paradigm (`s_axilite`): The data is accessed by the kernel through register interfaces and performed by software register reads/writes.

The Vitis kernel flow implements the following interfaces by default:

C-argument type	Paradigm	Interface protocol (I/O/Inout)
Scalar(pass by value)	Register	AXI4-Lite (<code>s_axilite</code>)
Array	Memory	AXI4 Memory Mapped (<code>m_axi</code>)
Pointer to array	Memory	<code>m_axi</code>
Pointer to scalar	Register	<code>s_axilite</code>
Reference	Register	<code>s_axilite</code>
<code>hls::stream</code>	Stream	AXI4-Stream (<code>axis</code>)

As you can see from the table above, a pointer to an array is implemented as an `m_axi` interface for data transfer, while a pointer to a scalar is implemented using the `s_axilite` interface. A scalar value passed as a constant does not need read access, while a pointer to a scalar value needs both read/write access. The `s_axilite` interface implements an additional internal protocol depending upon the C argument type. This internal implementation can be controlled using [Port-Level I/O Protocols](#). However, you should not modify the default port protocols in the Vitis kernel flow unless necessary.

Note: Vitis HLS will not automatically infer the default interfaces for the member elements of a struct/class when the elements require different interface types. For example, when one element of a struct requires a stream interface while another member element requires an `s_axilite` interface. You must explicitly define an INTERFACE pragma for each element of the struct instead of relying on the default interface assignment. If no INTERFACE pragma or directive is defined Vitis HLS will issue the following error message:

```
ERROR: [HLS 214-312] Vitis mode requires explicit INTERFACE pragmas for structs in the interface. Please add one INTERFACE pragma for each struct member field for argument 'd' of function 'dut(A&)' (example.cpp:19:0)
```

The default execution mode for Vitis kernel flow is pipelined execution, which enables overlapping execution of a kernel to improve throughput. This is specified by the `ap_ctrl_chain` block control protocol on the `s_axilite` interface.



TIP: The Vitis environment supports kernels with all of the supported block control protocols as described in [Block-Level Control Protocols](#).

The `vadd` function in the following code provides an example of interface synthesis.

```
#define VDATA_SIZE 16

typedef struct v_datatype { unsigned int data[VDATA_SIZE]; } v_dt;

extern "C" {
void vadd(const v_dt* in1, // Read-Only Vector 1
          const v_dt* in2, // Read-Only Vector 2
          v_dt* out_r, // Output Result for Addition
          const unsigned int size // Size in integer
) {
    unsigned int vSize = ((size - 1) / VDATA_SIZE) + 1;

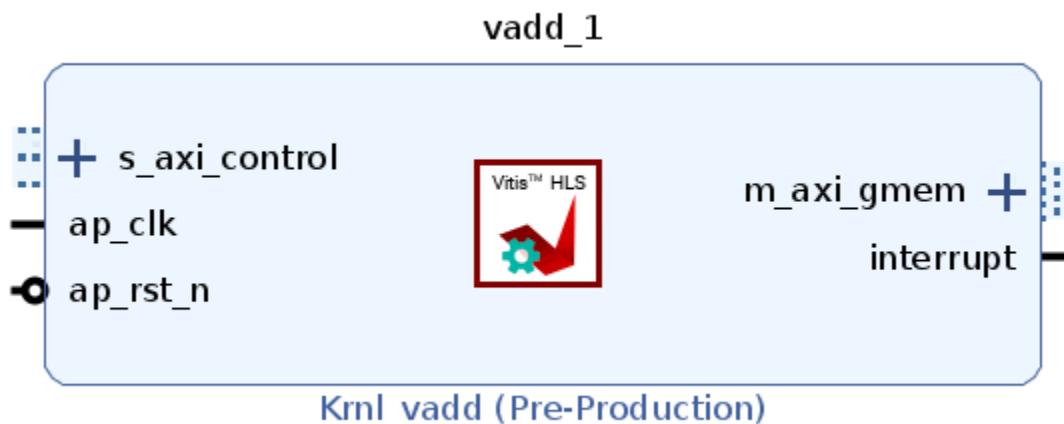
    // Auto-pipeline is going to apply pipeline to this loop
    vadd1:
    for (int i = 0; i < vSize; i++) {
        vadd2:
        for (int k = 0; k < VDATA_SIZE; k++) {
            out_r[i].data[k] = in1[i].data[k] + in2[i].data[k];
        }
    }
}
```

The `vadd` function includes:

- Two pointer inputs: `in1` and `in2`
- A pointer: `out_r` that the results are written to
- A scalar value `size`

With the default interface synthesis settings used by Vitis HLS for the Vitis kernel flow, the design is synthesized into an RTL block with the ports and interfaces shown in the following figure.

Figure 61: RTL Ports After Default Interface Synthesis



The tool creates three types of interface ports on the RTL design to handle the flow of both data and control.

- Clock, Reset, and Interrupt ports: `ap_clk` and `ap_rst_n` and `interrupt` are added to the kernel.
- AXI4-Lite interface: `s_axi_control` interface which contains the scalar arguments like `size`, and manages address offsets for the `m_axi` interface, and defines the block control protocol.
- AXI4 memory mapped interface: `m_axi_gmem` interface which contains the pointer arguments: `in1`, `in2`, and `out_r`

Details of M_AXI Interfaces for Vitis

AXI4 memory-mapped (`m_axi`) interfaces allow kernels to read and write data in global memory (DDR, HBM, PLRAM). Memory-mapped interfaces are a convenient way of sharing data across different elements of the accelerated application, such as between the host and kernel, or between kernels on the accelerator card. The main advantages for `m_axi` interfaces are listed below:

- The interface has independent read and write channels

- It supports burst-based accesses with potential performance of ~19 GB/s
- It provides a queue for outstanding transactions
- **Understanding Burst Access:** AXI4 memory-mapped interfaces support high throughput bursts of up to 4K bytes with just a single address phase. With burst mode transfers, Vitis HLS reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput. Burst mode of operation is possible when you use the C `memcpy` function or a pipelined `for` loop. Refer to [Controlling AXI4 Burst Behavior](#) or [AXI Burst Transfers](#) for more information.
- **Automatic Port Widening and Port Width Alignment:**

As discussed in [Automatic Port Width Resizing](#), Vitis HLS has the ability to automatically widen a port width to facilitate data transfers and improve burst access, if a burst access can be seen by the tool. Therefore all the preconditions needed for bursting, as described in [AXI Burst Transfers](#), are also needed for port resizing.

In the Vitis Kernel flow automatic port width resizing is enabled by default with the following configuration commands (notice that one command is specified as bits and the other is specified as bytes):

```
config_interface -m_axi_max_widen_bitwidth 512
config_interface -m_axi_alignment_byte_size 64
```

- **Rules for Offset:**



IMPORTANT! In the Vitis kernel flow the default mode of operation is `offset=direct` and `default_slave_interface=s_axilite` and should not be changed.

The correct specification of the offset will let the HLS kernel correctly integrate into the Vitis system. Refer to [Offset and Modes of Operation](#) for more information.

- **Bundle Interfaces - Performance vs. Resource Utilization:**

By default, Vitis HLS groups function arguments with compatible options into a single `m_axi` interface adapter as described in [M_AXI Bundles](#). Bundling ports into a single interface helps save device resources by eliminating AXI4 logic, which can be necessary when working in congested designs.

However, a single interface bundle can limit the performance of the kernel because all the memory transfers have to go through a single interface. The `m_axi` interface has independent READ and WRITE channels, so a single interface can read and write simultaneously, though only at one location. Using multiple bundles lets you increase the bandwidth and throughput of the kernel by creating multiple interfaces to connect to memory banks.

Details of S_AXILITE Interfaces for Vitis

In C++, a function starts to process data when the function is called from a parent function. The function call is pushed onto the stack when called, and removed from the stack when processing is complete to return control to the calling function. This process ensures the parent knows the status of the child.

Since the host and kernel occupy two separate compute spaces in the Vitis kernel flow, the "stack" is managed by the Xilinx Run Time (XRT), and communication is managed through the `s_axilite` interface. The kernel is software controlled through XRT by reading and writing the control registers of an `s_axilite` interface as described in [S_AXILITE Control Register Map](#). The interface provides the following features:

- **Control Protocols:** The block control protocol defines control registers in the `s_axilite` interface that let you set control signals to manage execution and operation of the kernel.
- **Scalar Arguments:** Scalar inputs on a kernel are typical, and can be thought of as programming constants or parameters. The host application transfers these values through the `s_axilite` interface.
- **Pointers to Scalar Arguments:** Vitis HLS lets you read to or write from a pointer to a scalar value when assigned to an `s_axilite` interface. Pointers are assigned by default to `m_axi` interfaces, so this requires you to manually assign the pointer to the `s_axilite` using the `INTERFACE` pragma or directive:

```
int top(int *a, int *b) {  
    #pragma HLS interface s_axilite port=a
```

- **Rules for Offset:**

Note: The Vitis kernel flow determines the required offsets. Do not specify the `offset` option in that flow.

- **Rules for Bundle:**

The Vitis kernel flow supports only a single `s_axilite` interface, which means that all `s_axilite` interfaces must be bundled together.

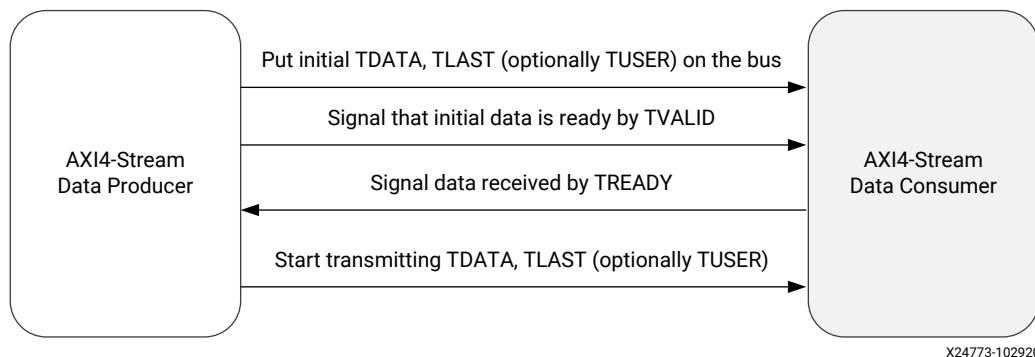
- When no bundle is specified the tool automatically creates a default bundle named `Control`.
- If for some reason you want to manually specify the bundle name, you must apply the same bundle to all `s_axilite` interfaces to create a single bundle.

Details of AXIS Interfaces for Vitis

The AXI4-Stream protocol (AXIS) defines a single uni-directional channel for streaming data in a sequential manner. The AXI4-Stream interfaces can burst an unlimited amount of data, which significantly improves performance. Unlike the AXI4 memory-mapped interface which needs an address to read/write the memory, the AXIS interface simply passes data to another AXIS interface without needing an address, and so uses fewer device resources. Combined, these features make the streaming interface a light-weight high performance interface.

The AXI4-Stream works on an industry-standard `ready/valid` handshake between a producer and consumer, as shown in the figure below. The data transfer is started once the producer sends the `TVALID` signal, and the consumer responds by sending the `TREADY` signal. This handshake of data and control should continue until either `TREADY` or `TVALID` are set low, or the producer asserts the `TLAST` signal indicating it is the last data packet of the transfer.

Figure 62: AXI4-Stream Handshake



IMPORTANT! The AXIS interface can only be assigned to the top-level arguments (ports) of a kernel or IP, and cannot be assigned to the arguments of functions internal to the design. Streaming channels used inside the HLS design should use `hls::stream` and not an AXIS interface.

You should define the streaming data type using `hls::stream<T_data_type>`, and use the `ap_axis` struct type to implement the AXIS interface. As explained in [AXI4-Stream Interfaces](#) the `ap_axis` struct lets you choose the implementation of the interface as with or without side-channels:

- [AXI4-Stream Interfaces without Side-Channels](#) implements the AXIS interface as a very light-weight interface using fewer resources
- [AXI4-Stream Interfaces with Side-Channels](#) implements a full featured interface providing greater control

TIP: You should not define your own struct for modeling the AXIS signals (side channels, `TLAST`, `TVALID`). Instead you can overload the `TDATA` signal for implementing your data type .

Interfaces for Vivado IP Flow

The Vivado IP flow supports a wide variety of I/O protocols and handshakes due to the requirement of supporting FPGA design for a wide variety of applications. This flow supports a traditional system design flow where multiple IP are integrated into a system. IP can be generated through Vitis HLS. In this IP flow there are two modes of control for execution of the system:

- Software Control: The system is controlled through a software application running on an embedded Arm processor or external x86 processor, using drivers to access elements of the hardware design, and reading and writing registers in the hardware to control the execution of IP in the system.
- Self Synchronous: In this mode the IP exposes signals which are used for starting and stopping the kernel. These signals are driven by other IP or other elements of the system design that handles the execution of the IP.

The Vivado IP flow supports memory, stream, and register interface paradigms where each paradigm supports different interface protocols to communicate with the external world, as shown in the following table. Note that while the Vitis kernel flow supports only the AXI4 interface adapters, this flow supports a number of different interface types.

Paradigm	Description	s
Memory	Data is accessed by the kernel through memory such as DDR, HBM, PLRAM/BRAM/URAMSupported Interface Protocol	ap_memory, BRAM, AXI4 Memory Mapped (m_axi)
Stream	Supported InterfaceData is streamed into the kernel from another streaming source, such as video processor or another kernel, and can also be streamed out of the kernel.	ap_fifo, AXI4-Stream (axis)
Register	Data is accessed by the kernel through register interfaces performed by register reads and writes.	ap_none, ap_hs, ap_ack, ap_ovld, ap_vld, and AXI4-Lite adapter (s_axilite).

The default interfaces are defined by the C-argument type in the top-level function, and the default paradigm, as shown in the following table.

C-Argument Type	Supported Paradigms	Default Paradigm	Default Interface Protocol		
			Input	Output	Inout
Scalar variable (pass by value)	Register	Register	ap_none	N/A	N/A
Array	Memory, Stream	Memory	ap_memory	ap_memory	ap_memory
Pointer	Memory, Stream, Register	Register	ap_none	ap_vld	ap_ovld
Reference	Register	Register	ap_none	ap_vld	ap_vld
hls::stream	Stream	Stream	ap_fifo	ap_fifo	N/A

The default execution mode for Vivado IP flow is sequential execution, which requires the HLS IP to complete one iteration before starting the next. This is specified by the `ap_ctrl_hs` block control protocol. The control protocol can be changed as specified in [Block-Level Control Protocols](#).

The `vadd` function in the following code provides an example of interface synthesis in the Vivado IP flow.

```
#define VDATA_SIZE 16

typedef struct v_datatype { unsigned int data[VDATA_SIZE]; } v_dt;

extern "C" {
void vadd(const v_dt* in1, // Read-Only Vector 1
          const v_dt* in2, // Read-Only Vector 2
          v_dt* out_r, // Output Result for Addition
          const unsigned int size // Size in integer
) {

    unsigned int vSize = ((size - 1) / VDATA_SIZE) + 1;

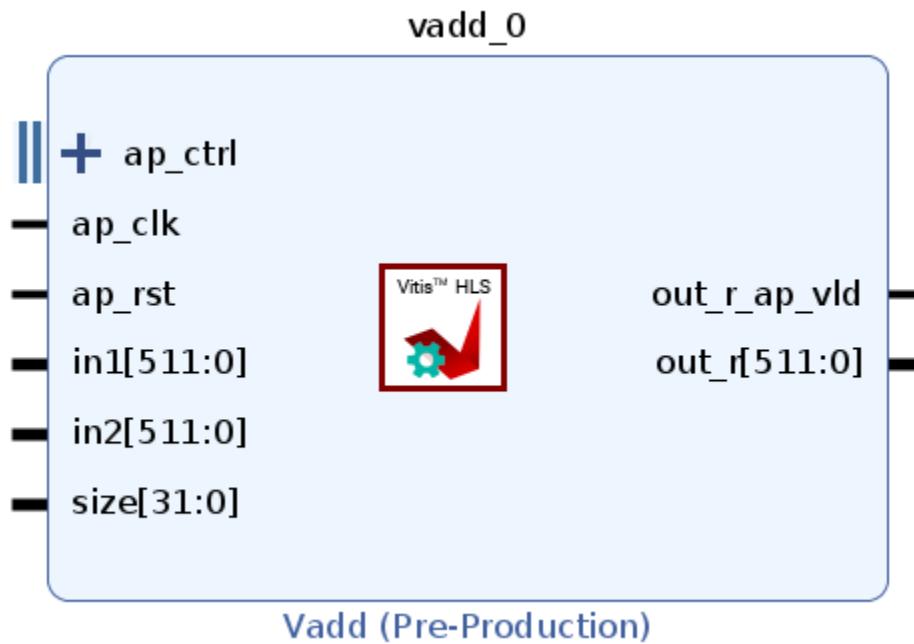
    // Auto-pipeline is going to apply pipeline to this loop
    vadd1:
    for (int i = 0; i < vSize; i++) {
        vadd2:
        for (int k = 0; k < VDATA_SIZE; k++) {
            out_r[i].data[k] = in1[i].data[k] + in2[i].data[k];
        }
    }
}
}
```

The `vadd` function includes:

- Two pointer inputs: `in1` and `in2`
- A pointer: `out_r` that the results are written to
- A scalar value `size`

With the default interface synthesis settings used for the Vivado IP flow, the design is synthesized into an RTL block with the ports and interfaces shown in the following figure.

Figure 63: RTL Ports After Default Interface Synthesis



In the default Vivado IP flow the tool creates three types of interface ports on the RTL design to handle the flow of both data and control.

- Clock and Reset ports: `ap_clk` and `ap_rst` are added to the kernel.
- Block-level control protocol: The `ap_ctrl` interface is implemented as an `s_axilite` interface.
- Port-level interface protocols: These are created for each argument in the top-level function and the function return (if the function returns a value). As explained in the table above most of the arguments use a port protocol of `ap_none`, and so have no control signals. In the `vadd` example above these ports include: `in1`, `in2`, and `size`. However, the `out_r_o` output port uses the `ap_vld` protocol and so is associated with the `out_r_o_ap_vld` signal.

AP_Memory in the Vivado IP Flow

The `ap_memory` is the default interface for the memory paradigm described in the tables above. In the Vivado IP flow it is used for communicating with memory resources such as BRAM and URAM. The `ap_memory` protocol also follows the address and data phase. The protocol initially requests to read/write the resource and waits until it receives an acknowledgment of the resource availability. It then initiates the data transfer phase of read/write.

An important consideration for `ap_memory` is that it can only perform a single beat data transfer to a single address, which is different from `m_axi` which can do burst accesses. This makes the `ap_memory` a lightweight protocol, compared to the others.

- **Memory Resources:** By default Vitis HLS implements a protocol to communicate with a single-port RAM resource. You can control the implementation of the protocol by specifying the `storage_type` as part of the INTERFACE pragma or directive. The `storage_type` lets you explicitly define which type of RAM is used, and which RAM ports are created (single-port or dual-port). If no `storage_type` is specified Vitis HLS uses:
 - A single-port RAM by default.
 - A dual-port RAM if it reduces the initiation interval or latency.

M_AXI Interfaces in the Vivado IP Flow

AXI4 memory-mapped (`m_axi`) interfaces allow an IP to read and write data in global memory (DDR, HBM, PLRAM). Memory-mapped interfaces are a convenient way of sharing data across multiple IP. The main advantages for `m_axi` interfaces are listed below:

- The interface has independent read and write channels
- It supports burst-based accesses with potential performance of ~19 GB/s
- It provides a queue for outstanding transactions
- **Understanding Burst Access:** AXI4 memory-mapped interfaces support high throughput bursts of up to 4K bytes with just a single address phase. With burst mode transfers, Vitis HLS reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput. Burst mode of operation is possible when you use the C `memcpy` function or a pipelined `for` loop. Refer to [Controlling AXI4 Burst Behavior](#) or [AXI Burst Transfers](#) for more information.
- **Automatic Port Widening and Port Width Alignment:**

As discussed in [Automatic Port Width Resizing](#), Vitis HLS has the ability to automatically widen a port width to facilitate data transfers and improve burst access when all the preconditions needed for bursting are present. In the Vivado IP flow the following configuration settings disable automatic port width resizing by default. To enable this feature you must change these configuration options (notice that one command is specified as bits and the other is specified as bytes):

```
config_interface -m_axi_max_widen_bitwidth 0
config_interface -m_axi_alignment_byte_size 0
```

- **Specifying Alignment for Vivado IP mode:**

The alignment for an `m_axi` port allows the port to read and write memory according to the specified alignment. Choosing the correct alignment is important as it will impact performance in the best case, and can impact functionality in the worst case.

Aligned memory access means that the pointer (or the start address of the data) is a multiple of a type-specific value called the alignment. The alignment is the natural address multiple where the type must be or should be stored (e.g. for performance reasons) on a Memory. For example, Intel 32-bit architecture stores words of 32 bits, each of 4 bytes in the memory. The data is aligned to one-word or 4-byte boundary.

The alignment should be consistent in the system. The alignment is determined when the IP is operating in AXI4 master mode and should be specified, like the Intel 32-bit architecture with 4-byte alignment. When the IP is operating in slave mode the alignment should match the alignment of the master.

- **Rules for Offset:**

The default for `m_axi` offset is `offset=direct` and `default_slave_interface=s_axilite`. However, in the Vivado IP flow you can change it as described in [Offset and Modes of Operation](#).

- **Bundle Interfaces - Performance vs. Resource Utilization:**

By default, Vitis HLS groups function arguments with compatible options into a single `m_axi` interface adapter as described in [M_AXI Bundles](#). Bundling ports into a single interface helps save device resources by eliminating AXI4 logic, which can be necessary when working in congested designs.

However, a single interface bundle can limit the performance of the IP because all the memory transfers have to go through a single interface. The `m_axi` interface has independent READ and WRITE channels, so a single interface can read and write simultaneously, though only at one location. Using multiple bundles lets you increase performance by creating multiple interfaces to connect to memory banks.

S_AXILITE in the Vivado IP Flow

In the Vivado IP flow, the default execution control is managed by register reads and writes through an `s_axilite` interface using the default `ap_ctrl_hs` control protocol. The IP is software controlled by reading and writing the control registers of an `s_axilite` interface as described in [S_AXILITE Control Register Map](#).

The `s_axilite` interface provides the following features:

- **Control Protocols:** The block control protocol as specified in [Block-Level Control Protocols](#).
- **Scalar Arguments:** Scalar arguments from the top-level function can be mapped to an `s_axilite` interface which creates a register for the value as described in [S_AXILITE Control Register Map](#). The software can perform reads/writes to this register space.
- **Rules for Offset:** The Vivado IP flow defines the size, or range of addresses assigned to a port based on the data type of the associated C-argument in the top-level function. However, the tool also lets you manually define the offset size as described in [S_AXILITE Offset Option](#).

- **Rules for Bundle:** In the Vivado IP flow you can specify multiple bundles using the `s_axilite` interface, and this will create a separate interface adapter for each bundle you have defined. However, there are some rules related to using multiple bundles that you should be familiar with as explained in [S_AXILITE Bundle Rules](#).

AP_FIFO in the Vivado IP Flow

In the Vivado IP flow, the `ap_fifo` interface protocol is the default interface for the streaming paradigm on the interface for communication with a memory resource FIFO, and can also be used as a communication channel between different functions inside the IP. This protocol should only be used if the data is accessed sequentially, and Xilinx *strongly recommends* using the `hls::stream<data_type>` which implements a FIFO.



TIP: The `<data_type>` should not be the same as the `T_data_type`, which should only be used on the interface.

AXIS Interfaces in the Vivado IP Flow

The AXI4-Stream protocol (`axis`) is an alternative for streaming interfaces, and defines a single uni-directional channel for streaming data in a sequential manner. Unlike the `m_axi` protocol, the AXI4-Stream interfaces can burst an unlimited amount of data, which significantly improves performance. Unlike the AXI4 memory-mapped interface which needs an address to read/write the memory, the `axis` interface simply passes data to another `axis` interface without needing an address, and so uses fewer device resources. Combined, these features make the streaming interface a light-weight high performance interface as described in [AXI4-Stream Interfaces](#).

AXI Adapter Interface Protocols



IMPORTANT! As discussed in [Interfaces for Vitis Kernel Flow](#), the AXI4 adapter interfaces are the default interfaces used by Vitis HLS for the Vitis Application Acceleration Development flow, though they are also supported in the Vivado IP flow. The AXI4-Stream Accelerator Adapter is a soft Xilinx® LogiCORE™ Intellectual Property (IP) core used as a infrastructure block for connecting hardware accelerators to embedded CPUs.

The AXI4 interfaces supported by Vitis HLS include the AXI4-Stream interface (`axis`), AXI4-Lite (`s_axilite`), and AXI4 master (`m_axi`) interfaces. For a complete description of the AXI4 interfaces, including timing and ports, see the [Vivado Design Suite: AXI Reference Guide \(UG1037\)](#).

- **`m_axi`:** Specify on arrays and pointers (and references in C++) only. The `m_axi` mode specifies an [AXI4 Memory Mapped interface](#).



TIP: You can group bundle arguments into a single `m_axi` interface.

- **s_axilite**: Specify this protocol on any type of argument except streams. The `s_axilite` mode specifies an [AXI4-Lite slave interface](#).



TIP: You can bundle multiple arguments into a single `s_axilite` interface.

- **axis**: Specify this protocol on input arguments or output arguments only, not on input/output arguments. The `axis` mode specifies an [AXI4-Stream interface](#).

AXI4 Master Interface

AXI4 memory-mapped (`m_axi`) interfaces allow kernels to read and write data in global memory (DDR, HBM, PLRAM). Memory-mapped interfaces are a convenient way of sharing data across different elements of the accelerated application, such as between the host and kernel, or between kernels on the accelerator card. The main advantages for `m_axi` interfaces are listed below:

- The interface has a separate and independent read and write channels
- It supports burst-based accesses with potential performance of ~17 GB/s
- It provides support for outstanding transactions

In the Vitis Kernel flow the `m_axi` interface is assigned by default to pointer and array arguments. In this flow it supports the following default features:

- Pointer and array arguments are automatically mapped to the `m_axi` interface
- The default mode of operation is `offset=slave` in the Vitis flow and should not be changed
- All pointer and array arguments are mapped to a single interface bundle to conserve device resources, and ports share read and write access across the time it is active
- The default alignment in the Vitis flow is set to 64 bytes
- The maximum read/write burst length is set to 16 by default

While not used by default in the Vivado IP flow, when the `m_axi` interface is specified it has the following default features:

- The default operation mode is `offset=off` but you can change it as described in [Offset and Modes of Operation](#)
- Assigned pointer and array arguments are mapped to a single interface bundle to conserve device resources, and share the interface across the time it is active
- The default alignment in Vivado IP flow is set to 1 byte
- The maximum read/write burst length is set to 16 by default

In both the Vivado IP flow and Vitis kernel flow, the INTERFACE pragma or directive can be used to modify default values as needed. Some customization can help improve design performance as described in [Optimizing AXI System Performance](#).

You can use an AXI4 master interface on array or pointer/reference arguments, which Vitis HLS implements in one of the following modes:

- Individual data transfers
- Burst mode data transfers

With individual data transfers, Vitis HLS reads or writes a single element of data for each address. The following example shows a single read and single write operation. In this example, Vitis HLS generates an address on the AXI interface to read a single data value and an address to write a single data value. The interface transfers one data value per address.

```
void bus (int *d) {
    static int acc = 0;

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

With burst mode transfers, Vitis HLS reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput. Burst mode of operation is possible when you use the C `memcpy` function or a pipelined `for` loop. Refer to [AXI Burst Transfers](#) for more information.



IMPORTANT! The C `memcpy` function is only supported for synthesis when used to transfer data to or from a top-level function argument specified with an AXI4 master interface.

The following example shows a copy of burst mode using the `memcpy` function. The top-level function argument `a` is specified as an AXI4 master interface.

```
void example(volatile int *a){

    //Port a is assigned to an AXI4 master interface
    #pragma HLS INTERFACE mode=m_axi depth=50 port=a
    #pragma HLS INTERFACE mode=s_axilite port=return

    int i;
    int buff[50];

    //memcpy creates a burst access to memory
    memcpy(buff,(const int*)a,50*sizeof(int));

    for(i=0; i < 50; i++){
        buff[i] = buff[i] + 100;
    }

    memcpy((int *)a,buff,50*sizeof(int));
}
```

When this example is synthesized, it results in the interface shown in the following figure.

Note: In this figure, the AXI4 interfaces are collapsed.

Figure 64: AXI4 Interface



The following example shows the same code as the preceding example but uses a `for` loop to copy the data out:

```
void example(volatile int *a){

#pragma HLS INTERFACE mode=m_axi depth=50 port=a
#pragma HLS INTERFACE mode=s_axilite port=return

//Port a is assigned to an AXI4 master interface

int i;
int buff[50];

//memcpy creates a burst access to memory
memcpy(buff,(const int*)a,50*sizeof(int));

for(i=0; i < 50; i++){
    buff[i] = buff[i] + 100;
}

for(i=0; i < 50; i++){
#pragma HLS PIPELINE
    a[i] = buff[i];
}
}
```

When using a `for` loop to implement burst reads or writes, follow these requirements:

- Pipeline the loop
- Access addresses in increasing order
- Do not place accesses inside a conditional statement
- For nested loops, do not flatten loops, because this inhibits the burst operation

Note: Only one read and one write is allowed in a `for` loop unless the ports are bundled in different AXI ports. The following example shows how to perform two reads in burst mode using different AXI interfaces.

In the following example, Vitis HLS implements the port reads as burst transfers. Port `a` is specified without using the `bundle` option and is implemented in the default AXI interface. Port `b` is specified using a named bundle and is implemented in a separate AXI interface called `d2_port`.

```
void example(volatile int *a, int *b){

#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE mode=m_axi depth=50 port=a
#pragma HLS INTERFACE mode=m_axi depth=50 port=b bundle=d2_port

int i;
int buff[50];

//copy data in
for(i=0; i < 50; i++){
#pragma HLS PIPELINE
buff[i] = a[i] + b[i];
}
...
}
```

Offset and Modes of Operation



IMPORTANT! In the Vitis kernel flow the default mode of operation is `offset=slave` and should not be changed.

The AXI4 Master interface has a read/write address channel that can be used to read/write specific addresses. By default the `m_axi` interface starts all read and write operations from the address `0x00000000`. For example, given the following code, the design reads data from addresses `0x00000000` to `0x000000C7` (50 32-bit words, gives 200 bytes), which represents 50 address values. The design then writes data back to the same addresses.

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

void example(volatile int *a){

#pragma HLS INTERFACE mode=m_axi port=a depth=50

int i;
int buff[50];

//memcpy creates a burst access to memory
//multiple calls of memcpy cannot be pipelined and will be scheduled
sequentially
//memcpy requires a local buffer to store the results of the memory
transaction
memcpy(buff,(const int*)a,50*sizeof(int));

for(i=0; i < 50; i++) {
```

```

        buff[i] = buff[i] + 100;
    }

    memcpy((int *)a, buff, 50*sizeof(int));
}

```

The tool provides the capability to let the base address be configured statically in the Vivado IP for instance, or dynamically by the application or another IP during run time.

The `m_axi` interface can be both a master initiating transactions, and also a slave interface that receives the data and sends acknowledgment. Depending on the mode specified with the `offset` option of the INTERFACE pragma, an HLS IP can use multiple approaches to set the base address.



TIP: The `config_interface -m_axi_offset` command provides a global setting for the offset, that can be overridden for specific `m_axi` interfaces using the INTERFACE pragma `offset` option.

- **Master Mode:** When acting as a master interface with different `offset` options, the `m_axi` interface start address can be either hard-coded or set at run time.
 - `offset=off`: Vitis HLS sets a base address for the `m_axi` interface when the IP is used in the Vivado IP integrator tool. One disadvantage with this approach is that you cannot change the base address during run time. See [Customizing AXI4 Master Interfaces in IP Integrator](#) for setting the base address.

The following example is synthesized with `offset=off`, the default for the Vivado IP flow.

```

void example(volatile int *a){
#pragma HLS INTERFACE m_axi depth=50 port=a offset=off

    int i;
    int buff[50];

    //memcpy creates a burst access to memory
    //multiple calls of memcpy cannot be pipelined and will be scheduled
    //sequentially
    //memcpy requires a local buffer to store the results of the memory
    //transaction
    memcpy(buff,(const int*)a,50*sizeof(int));

    for(i=0; i < 50; i++){
        buff[i] = buff[i] + 100;
    }

    memcpy((int *)a,buff,50*sizeof(int));
}

```

- `offset=direct`: Vitis HLS generates a port on the IP for setting the address. Note the addition of the `a` port as shown in the figure below. This lets you update the address at run time, so you can have one `m_axi` interface reading and writing different locations. For example, an HLS module that reads data from an ADC into RAM, and an HLS module that processes that data. Since you can change the address on the module, while one HLS module is processing the initial dataset the other module can be reading more data into different address.

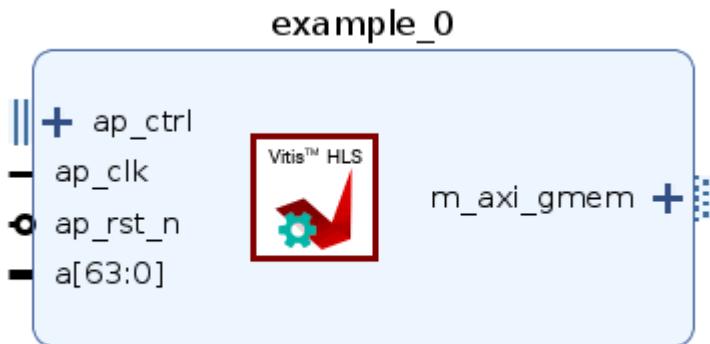
```
void example(volatile int *a){  

#pragma HLS INTERFACE m_axi depth=50 port=a offset=direct  

...  

}
```

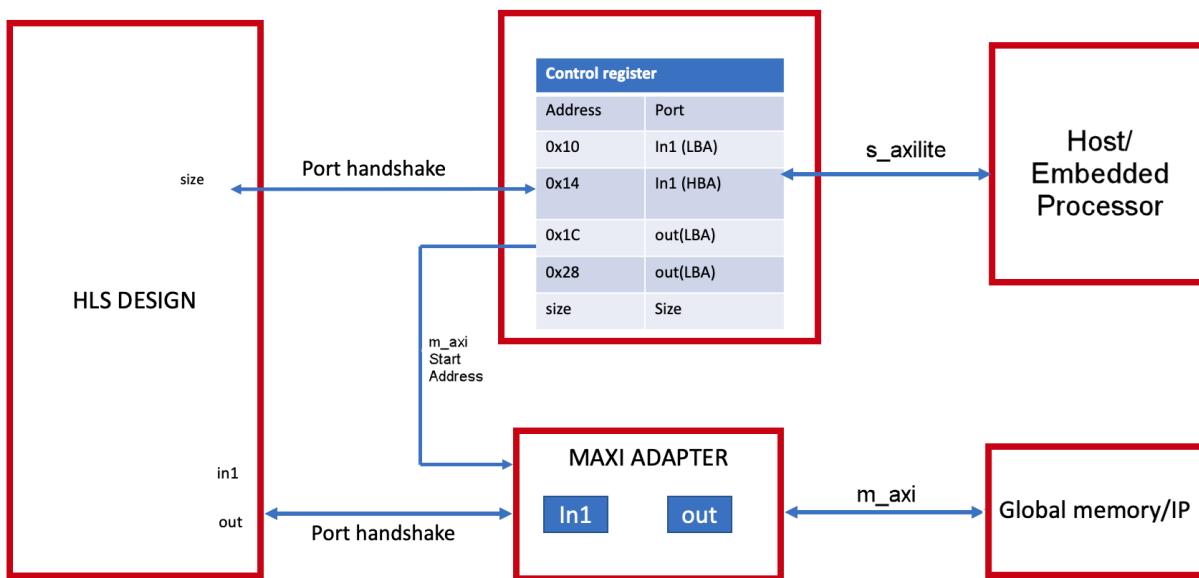
Figure 65: `offset=direct`



- *Slave Mode*: The slave mode for an interface is set with `offset=slave`. In this mode the IP will be controlled by the host application, or the micro-controller through the `s_axilite` interface. This is the default for the Vitis kernel flow, and can also be used in the Vivado IP flow. Here is the flow of operation:
 1. initially, the Host/CPU will start the IP or kernel using the block-level control protocol which is mapped to the `s_axilite` adapter.
 2. The host will send the scalars and address offsets for the `m_axi` interfaces through the `s_axilite` adapter.
 3. The `m_axi` adapter will read the start address from the `s_axilite` adapter and store it in a queue.
 4. The HLS design starts to read the data from the global memory.

As shown in the figure below, the HLS design will have both the `s_axilite` adapter for the base address, and the `m_axi` to perform read and write transfer to the global memory.

Figure 66: AXI Adapters in Slave Mode



Offset Rules

The following are rules associated with the `offset` option:

- Fully Specified Offset: When the user explicitly sets the offset value the tool uses the specified settings. The user can also set different offset values for different `m_axi` interfaces in the design, and the tool will use the specified offsets.

```
#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=out offset=direct
#pragma HLS INTERFACE mode=m_axi bundle=BUS_B port=in1 offset=slave
#pragma HLS INTERFACE mode=m_axi bundle=BUS_C port=in2 offset=off
```

- No Offset Specified: If there are no offsets specified in the INTERFACE pragma, the tool will defer to the setting specified by `config_interface -m_axi_offset`.

Note: If the global `m_axi_offset` setting is specified, and the design has an `s_axilite` interface, the global setting is ignored and `offset=slave` is assumed.

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

Controlling the Address Offset in an AXI4 Interface

By default, the AXI4 master interface starts all read and write operations from address 0x00000000. For example, given the following code, the design reads data from addresses 0x00000000 to 0x000000C7 (50 32-bit words, gives 200 bytes), which represents 50 address values. The design then writes data back to the same addresses.

```
void example(volatile int *a){

#pragma HLS INTERFACE mode=m_axi depth=50 port=a
#pragma HLS INTERFACE mode=s_axilite port=return bundle=AXILiteS

int i;
int buff[50];

memcpy(buff,(const int*)a,50*sizeof(int));

for(i=0; i < 50; i++){
    buff[i] = buff[i] + 100;
}
memcpy((int *)a,buff,50*sizeof(int));
}
```

To apply an address offset, use the `-offset` option with the INTERFACE directive, and specify one of the following options:

- `off`: Does not apply an offset address. This is the default.
- `direct`: Adds a 32-bit port to the design for applying an address offset.
- `slave`: Adds a 32-bit register inside the AXI4-Lite interface for applying an address offset.

In the final RTL, Vitis HLS applies the address offset directly to any read or write address generated by the AXI4 master interface. This allows the design to access any address location in the system.

If you use the `slave` option in an AXI interface, you must use an AXI4-Lite port on the design interface. Xilinx recommends that you implement the AXI4-Lite interface using the following pragma:

```
#pragma HLS INTERFACE mode=s_axilite port=return
```

In addition, if you use the `slave` option and you used several AXI4-Lite interfaces, you must ensure that the AXI master port offset register is bundled into the correct AXI4-Lite interface. In the following example, port `a` is implemented as an AXI master interface with an offset and AXI4-Lite interfaces called `AXI_Lite_1` and `AXI_Lite_2`:

```
#pragma HLS INTERFACE mode=m_axi port=a depth=50 offset=slave
#pragma HLS INTERFACE mode=s_axilite port=return bundle=AXI_Lite_1
#pragma HLS INTERFACE mode=s_axilite port=b bundle=AXI_Lite_2
```

The following INTERFACE directive is required to ensure that the offset register for port `a` is bundled into the AXI4-Lite interface called `AXI_Lite_1`:

```
#pragma HLS INTERFACE mode=s_axilite port=a bundle=AXI_Lite_1
```

M_AXI Bundles

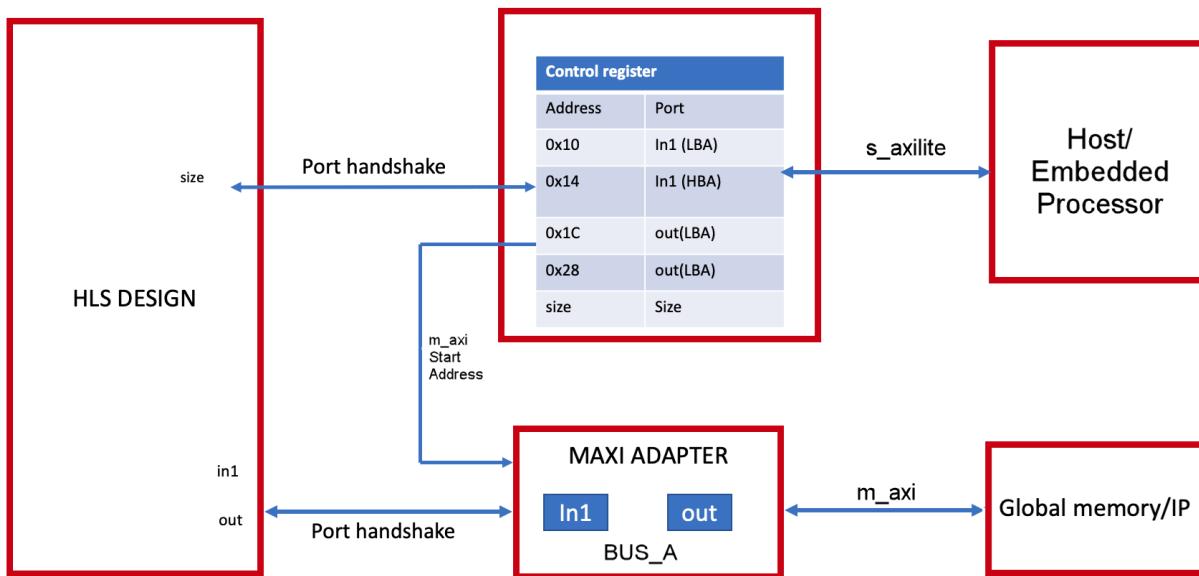
Vitis HLS groups function arguments with compatible options into a single `m_axi` interface adapter. Bundling ports into a single interface helps save FPGA resources by eliminating AXI logic, but it can limit the performance of the kernel because all the memory transfers have to go through a single interface. The `m_axi` interface has independent READ and WRITE channels, so a single interface can read and write simultaneously, though only at one location. Using multiple bundles the bandwidth and throughput of the kernel can be increased by creating multiple interfaces to connect to multiple memory banks.

In the following example all the pointer arguments are grouped into a single `m_axi` adapter using the interface option `bundle=BUS_A`, and adds a single `s_axilite` adapter for the `m_axi` offsets, the scalar argument `size`, and the function return.

```
extern "C" {
void vadd(const unsigned int *in1, // Read-Only Vector 1
          const unsigned int *in2, // Read-Only Vector 2
          unsigned int *out,      // Output Result
          int size                // Size in integer
) {

#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=out
#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=in1
#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=in2
#pragma HLS INTERFACE mode=s_axilite port=in1
#pragma HLS INTERFACE mode=s_axilite port=in2
#pragma HLS INTERFACE mode=s_axilite port=out
#pragma HLS INTERFACE mode=s_axilite port=size
#pragma HLS INTERFACE mode=s_axilite port=return
```

Figure 67: MAXI and S_AXILITE



You can also choose to bundle function arguments into separate interface adapters as shown in the following code. Here the argument `in2` is grouped into a separate interface adapter with `bundle=BUS_B`. This creates a new `m_axi` interface adapter for port `in2`.

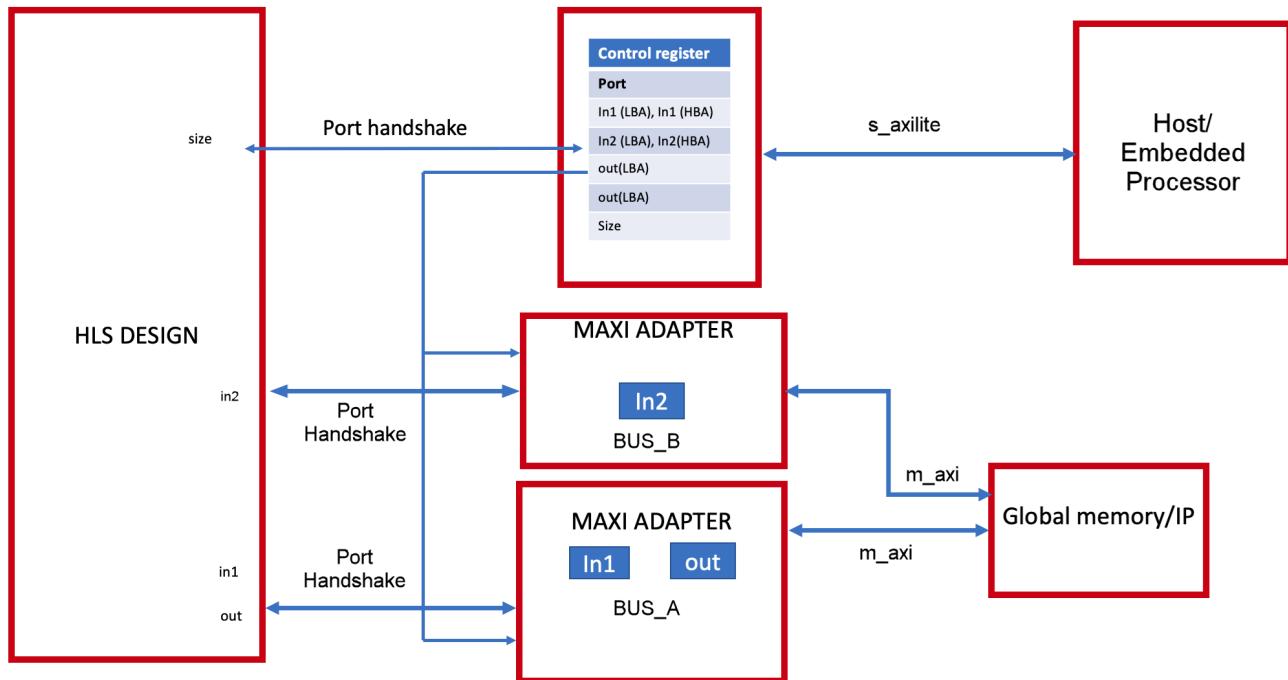
```

extern "C" {
void vadd(const unsigned int *in1, // Read-Only Vector 1
          const unsigned int *in2, // Read-Only Vector 2
          unsigned int *out,      // Output Result
          int size,              // Size in integer
          ) {

#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=out
#pragma HLS INTERFACE mode=m_axi bundle=BUS_A port=in1
#pragma HLS INTERFACE mode=m_axi bundle=BUS_B port=in2
#pragma HLS INTERFACE mode=m_axi port=in1
#pragma HLS INTERFACE mode=s_axilite port=in2
#pragma HLS INTERFACE mode=s_axilite port=out
#pragma HLS INTERFACE mode=s_axilite port=size
#pragma HLS INTERFACE mode=s_axilite port=return
}

```

Figure 68: 2 MAXI Bundles



Bundle Rules

The global configuration command `config_interface -m_axi_auto_max_ports false` will limit the number of interface bundles to the minimum required. It will allow the tool to group compatible ports into a single `m_axi` interface. The default setting for this command is disabled (false), but you can enable it to maximize bandwidth by creating a separate `m_axi` adapter for each port.

With `m_axi_auto_max_ports` disabled, the following are some rules for how the tool handles bundles under different circumstances:

- Default Bundle Name:** The tool groups all interface ports with no bundle name into a single `m_axi` interface port using the tool default name `bundle=<default>`, and names the RTL port `m_axi_<default>`. The following pragmas:

```
#pragma HLS INTERFACE mode=m_axi port=a depth=50
#pragma HLS INTERFACE mode=m_axi port=a depth=50
#pragma HLS INTERFACE mode=m_axi port=a depth=50
```

Result in the following messages:

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

2. User-Specified Bundle Names: The tool groups all interface ports with the same user-specified `bundle=<string>` into the same `m_axi` interface port, and names the RTL port the value specified by `m_axi_<string>`. Ports without bundle assignments are grouped into the default bundle as described above. The following pragmas:

```
#pragma HLS INTERFACE mode=m_axi port=a depth=50 bundle=BUS_A
#pragma HLS INTERFACE mode=m_axi port=b depth=50
#pragma HLS INTERFACE mode=m_axi port=c depth=50
```

Result in the following messages:

```
INFO: [RTGEN 206-500] Setting interface mode on port 'example/BUS_A' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem' to
'm_axi'.
INFO: [RTGEN 206-500] Setting interface mode on port 'example/gmem' to
'm_axi'.
```



IMPORTANT! If you bundle incompatible interfaces Vitis HLS issues a message and ignores the bundle assignment.

Controlling AXI4 Burst Behavior

An optimal AXI4 interface is one in which the design never stalls while waiting to access the bus, and after bus access is granted, the bus never stalls while waiting for the design to read/write. To create the optimal AXI4 interface, the following options are provided in the INTERFACE pragma or directive to specify the behavior of the bursts and optimize the efficiency of the AXI4 interface. Refer to [AXI Burst Transfers](#) for more information on burst transfers.

Some of these options use internal storage to buffer data and may have an impact on area and resources:

- `latency`: Specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (`latency`) before the read or write is expected. If this figure is too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be granted but the bus may stall waiting on the design to start the access.
- `max_read_burst_length`: Specifies the maximum number of data values read during a burst transfer.
- `num_read_outstanding`: Specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size: `num_read_outstanding*max_read_burst_length*word_size`.

- `max_write_burst_length`: Specifies the maximum number of data values written during a burst transfer.
- `num_write_outstanding`: Specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size: `num_read_outstanding*max_read_burst_length*word_size`

The following example can be used to help explain these options:

```
#pragma HLS interface mode=m_axi port=input offset=slave bundle=gmem0
depth=1024*1024*16/(512/8)
latency=100
num_read_outstanding=32
num_write_outstanding=32
max_read_burst_length=16
max_write_burst_length=16
```

The interface is specified as having a latency of 100. Vitis HLS seeks to schedule the request for burst access 100 clock cycles before the design is ready to access the AXI4 bus. To further improve bus efficiency, the options `num_write_outstanding` and `num_read_outstanding` ensure the design contains enough buffering to store up to 32 read and write accesses. This allows the design to continue processing until the bus requests are serviced. Finally, the options `max_read_burst_length` and `max_write_burst_length` ensure the maximum burst size is 16 and that the AXI4 interface does not hold the bus for longer than this.

These options allow the behavior of the AXI4 interface to be optimized for the system in which it will operate. The efficiency of the operation does depend on these values being set accurately.

Automatic Port Width Resizing

In the Vitis tool flow Vitis HLS provides the ability to automatically re-size `m_axi` interface ports to 512-bits to improve burst access. However, automatic port width resizing only supports standard C data types and does not support non-aggregate types such as `ap_int`, `ap_uint`, `struct`, or `array`.



IMPORTANT! Structs on the interface prevent automatic widening of the port. You must break the struct into individual elements to enable this feature.

Vitis HLS controls automatic port width resizing using the following two commands:

- `config_interface -m_axi_max_widen_bitwidth <N>`: Directs the tool to automatically widen bursts on M-AXI interfaces up to the specified bitwidth. The value of `<N>` must be a power-of-two between 0 and 1024.
- `config_interface -m_axi_alignment_byte_size <N>`: Note that burst widening also requires strong alignment properties. Assume pointers that are mapped to `m_axi` interfaces are at least aligned to the provided width in bytes (power of two). This can help automatic burst widening.

In the Vitis Kernel flow automatic port width resizing is enabled by default with the following:

```
config_interface -m_axi_max_widen_bitwidth 512
config_interface -m_axi_alignment_byte_size 64
```

In the Vivado IP flow this feature is disabled by default:

```
config_interface -m_axi_max_widen_bitwidth 0
config_interface -m_axi_alignment_byte_size 0
```

Automatic port width resizing will only re-size the port if a burst access can be seen by the tool. Therefore all the preconditions needed for bursting, as described in [AXI Burst Transfers](#), are also needed for port resizing. These conditions include:

- Must be a monotonically increasing order of access (both in terms of the memory location being accessed as well as in time). You cannot access a memory location that is in between two previously accessed memory locations- aka no overlap.
- The access pattern from the global memory should be in sequential order, and with the following additional requirements:
 - The sequential accesses need to be on a non-vector type
 - The start of the sequential accesses needs to be aligned to the widen word size
 - The length of the sequential accesses needs to be divisible by the widen factor

The following code example is used in the calculations that follow:

```
vadd_pipeline:
    for (int i = 0; i < iterations; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_len/c_n max = c_len/c_n

    // Pipelining loops that access only one variable is the ideal way to
    // increase the global memory bandwidth.
    read_a:
        for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
        result[x] = a[i * N + x];
    }

    read_b:
        for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
        result[x] += b[i * N + x];
    }

    write_c:
        for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
```

```

        c[i * N + x] = result[x];
    }
}
}
}
}
```

The width of the automatic optimization for the code above is performed in three steps:

1. The tool checks for the number of access patterns in the read_a loop. There is one access during one loop iteration, so the optimization determines the interface bit-width as $32 = 32 * 1$ (bitwidth of the int variable * accesses).
2. The tool tries to reach the default max specified by the config_interface m_axi_max_widen_bitwidth 512, using the following expression terms:

```
length = (ceil((loop-bound of index inner loops) *
(loop-bound of index - outer loops)) * #(of access-patterns))
```

- In the above code, the outer loop is an imperfect loop so there will not be burst transfers on the outer-loop. Therefore the length will only include the inner-loop. Therefore the formula will be shortened to:

```
length = (ceil((loop-bound of index inner loops)) * #(of access-
patterns))
```

or: $length = ceil(128) * 32 = 4096$

3. Is the calculated length a power of 2? If Yes, then the length will be capped to the width specified by the m_axi_max_widen_bitwidth.

There are some pros and cons to using the automatic port width resizing which you should consider when using this feature. This feature improves the read latency from the DDR as the tool is reading a big vector, instead of the data type size. It also adds more resources as it needs to buffer the huge vector and shift the data accordingly to the data path size.

Creating an AXI4 Interface with 32-bit Address

By default, Vitis HLS implements the AXI4 port with a 64-bit address bus. However, some devices such as the Zynq-7000 have a 32 bit address bus. In this case you can implement the AXI4 interface with a 32-bit address bus by disabling the m_axi_addr64 interface configuration option as follows:

1. Select **Solution** → **Solution Settings**.
2. In the Solution Settings dialog box, click the **General** category, and **Edit** the existing config_interface command, or click **Add** to add one.
3. In the Edit or Add dialog box, select **config_interface**, and disable **m_axi_addr64**.



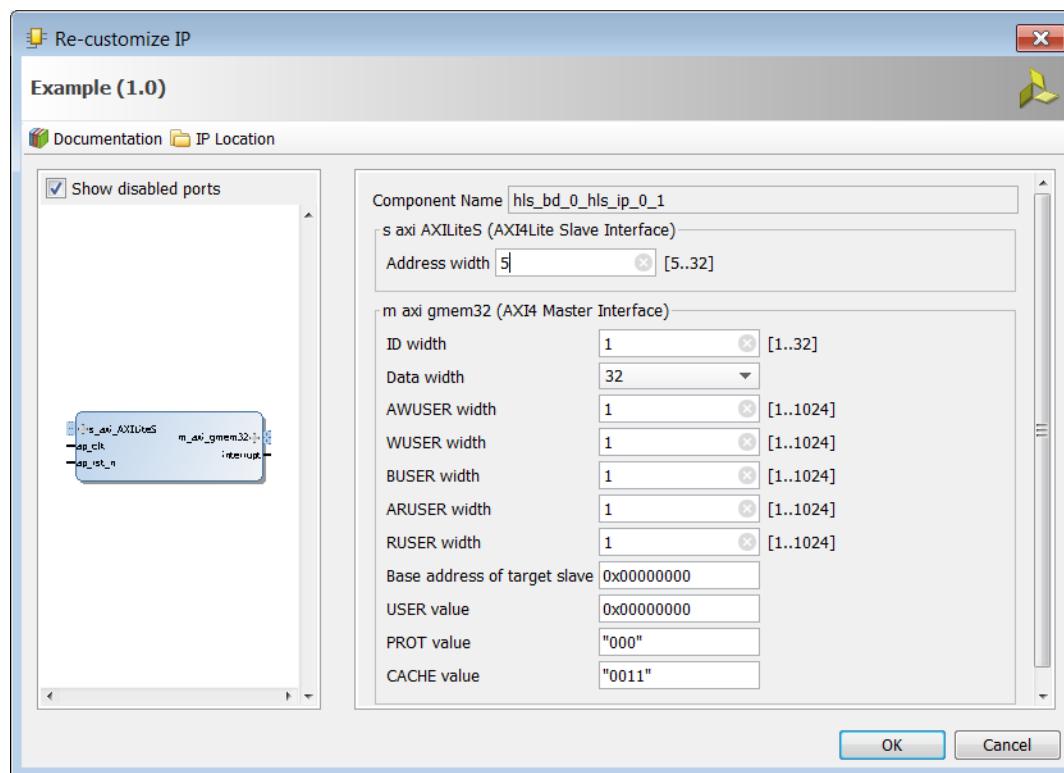
*IMPORTANT! When you disable the **m_axi_addr64** option, Vitis HLS implements all AXI4 interfaces in the design with a 32-bit address bus.*

Customizing AXI4 Master Interfaces in IP Integrator

When you incorporate an HLS RTL design that uses an AXI4 master interface into a design in the Vivado IP integrator, you can customize the block. From the block diagram in IP integrator, select the HLS block, right-click, and select **Customize Block** to customize any of the settings provided. A complete description of the AXI4 parameters is provided in this [link](#) in the Vivado Design Suite: AXI Reference Guide ([UG1037](#)).

The following figure shows the Re-Customize IP dialog box for the design shown below. This design includes an AXI4-Lite port.

Figure 69: Customizing AXI4 Master Interfaces in IP Integrator



AXI4-Lite Interface

Overview

An HLS IP or kernel can be controlled by a host application, or embedded processor using the Slave AXI4-Lite interface (`s_axilite`) which acts as a system bus for communication between the processor and the kernel. Using the `s_axilite` interface the host or an embedded processor can start and stop the kernel, and read or write data to it. When Vitis HLS synthesizes the design the `s_axilite` interface is implemented as an adapter that captures the data that was communicated from the host in registers on the adapter.

The AXI4-Lite interface performs several functions within a Vivado IP or Vitis kernel:

- It maps a block-level control mechanism which can be used to start and stop the kernel.
- It provides a channel for passing scalar arguments, pointers to scalar values, function return values, and address offsets for `m_axi` interfaces from the host to the IP or kernel
- For the Vitis Kernel flow:
 - The tool will automatically infer the `s_axilite` interface pragma to provide offsets to pointer arguments assigned to `m_axi` interfaces, scalar values, and function return type.
 - Vitis HLS lets you read to or write from a pointer to a scalar value when assigned to an `s_axilite` interface. Pointers are assigned by default to `m_axi` interfaces, so this requires you to manually assign the pointer to the `s_axilite` using the INTERFACE pragma or directive:

```
int top(int *a, int *b) {
#pragma HLS interface s_axilite port=a
```

- **Bundle:** Do not specify the `bundle` option for the `s_axilite` adapter in the Vitis Kernel flow. The tool will create a single `s_axilite` interface that will serve for the whole design.



IMPORTANT! HLS will return an error if multiple bundles are specified for the Vitis Kernel flow.

-
- **Offset:** The tool will automatically choose the offsets for the interface. Do not specify any offsets in this flow.
 - For the Vivado IP flow:
 - This flow will not use the `s_axilite` interface by default.
 - To use the `s_axilite` as a communication channel for scalar arguments, pointers to scalar values, offset to `m_axi` pointer address, and function return type, you must manually specify the INTERFACE pragma or directive.
 - **Bundle:** This flow supports multiple `s_axilite` interfaces, specified by bundle. Refer to [S_AXILITE Bundle Rules](#) for more information.
 - **Offset:** By default the tool will place the arguments in a sequential order starting from 0x10 in the control register map. Refer to [S_AXILITE Offset Option](#) for additional details.

S_AXILITE Example

The following example shows how Vitis HLS implements multiple arguments, including the function return, as an `s_axilite` interface. Because each pragma uses the same name for the `bundle` option, each of the ports is grouped into a single interface.

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE mode=s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=a bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=b bundle=BUS_A
```

```
#pragma HLS INTERFACE mode=s_axilite port=c bundle=BUS_A
#pragma HLS INTERFACE mode=ap_vld port=b

    *c += *a + *b;
}
```

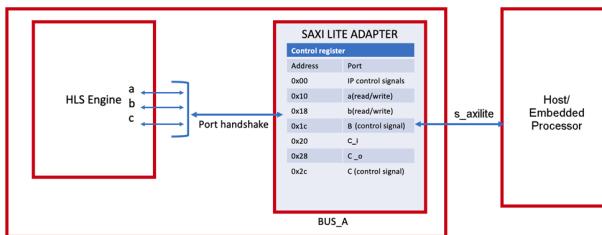


TIP: If you do not specify the `bundle` option, Vitis HLS groups all arguments into a single `s_axilite` bundle and automatically names the port.

The synthesized example will be part of a system that has three important elements as shown in the figure below:

1. Host application running on an x86 or embedded processor interacting with the IP or kernel
2. AXI Lite Adapter: The INTERFACE pragma implements an `s_axilite` adapter. The adapter has two primary functions: implementing the interface protocol to communicate with the host, and providing a Control Register Map to the IP or kernel.
3. The HLS engine or function that implements the design logic

Figure 70: S_AXILITE Adapter



By default, Vitis HLS automatically assigns the address for each port that is grouped into an `s_axilite` interface. The size, or range of addresses assigned to a port is dependent on the argument data type and the port protocol used, as described below. You can also explicitly define the address using the `offset` option as discussed in [S_AXILITE Offset Option](#).

- Port a: By default, is implemented as `ap_none`. 1-word for the data signal is assigned and only 3 bits are used as the argument data type is `char`. Remaining bits are unused.
- Port b: is implemented as `ap_vld` defined by the INTERFACE pragma in the example. The corresponding control register is of size 2 bytes (16-bits) and is divided into two sections as follows:
 - (0x1c) Control signal : 1-word for the control signal is assigned.
 - (0x18) Data signal: 1-word for the data signal is assigned and only 3 bits are used as the argument data type is `char`. Remaining bits are unused.
- Port c: By default, is implemented as `ap_ovld` as an output. The corresponding control register is of size 4 bytes (32 bits) and is divided into three sections:
 - (0x20) Data signal of `c_i`: 1-word for the input data signal is assigned, and only 3 bits are used as the argument data type is `char`, the rest are not used.

- (0x24) Reserved Space
- (0x28) Data signal of `c_o`: 1-word for the output data signal is assigned.
- (0x2c) Control signal of `c_o` : 1-word for control signal `ap_ovld` is assigned and only 3 bits are used as the argument data type is `char`. Remaining bits are unused.

In operation the host application will initially start the kernel by writing into the Control address space (0x00). The host/CPU completes the initial setup by writing into the other address spaces which are associated with the various function arguments as defined in the example.

The control signal for port b is asserted and only then can the HLS engine read ports a and b (port a is `ap_none` and does not have a control signal). Until that time the design is stalled and waiting for the *valid* register to be set for port b. Each time port b is read by the HLS engine the input *valid* register is cleared and the register resets to logic 0.

After the HLS engine finishes its computation, the output value on port C is stored in the control register and the corresponding *valid* bit is set for the host to read. After the host reads the data, the HLS engine will write the `ap_done` bit in the Control register (0x00) to mark the end of the IP computation.

Vitis HLS reports the assigned addresses in the [S_AXILITE Control Register Map](#), and also provides them in [C Driver Files](#) to aid in your software development. Using the `s_axilite` interface, you can output C driver files for use with code running on an embedded or x86 processor using provided C application program interface (API) functions, to let you control the hardware from your software.

S_AXILITE Control Register Map

Vitis HLS automatically generates a Control Register Map for controlling the Vivado IP or Vitis kernel, and the ports grouped into `s_axilite` interface. The register map, which is added to the generated RTL files, can be divided into two sections:

1. Block-level control signals
2. Function arguments mapped into the `s_axilite` interface

In the Vitis kernel flow, the block protocol is associated with the `s_axilite` interface by default. To change the default block protocol, specify the interface pragma as follows:

```
#pragma HLS INTERFACE mode=ap_ctrl_hs port=return
```

In the Vivado IP flow though, the block control protocol is assigned to its own interface, `ap_ctrl`, as seen in [Interfaces for Vivado IP Flow](#). However, if you are using an `s_axilite` interface in your IP, you can also assign the block control protocol to that interface using the following INTERFACE pragmas, as an example:

```
#pragma HLS INTERFACE mode=s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE mode=ap_ctrl_hs port=return bundle=BUS_A
```

In the Control Register Map, Vitis HLS reserves addresses `0x00` through `0x0C` for the block-level protocol signals and interrupt controls, as shown below:

Address	Description
0x00	Control signals
0x04	Global Interrupt Enable Register
0x08	IP Interrupt Enable Register (Read/Write)
0x0c	IP Interrupt Status Register (Read/TOW)

The Control signals (`0X00`) contains `ap_start`, `ap_done`, `ap_ready`, and `ap_idle`; and in the case of `ap_ctrl_chain` the block protocol also contains `ap_continue`. These are the *block-level interface* signals which are accessed through the `s_axilite` adapter.

To start the block operation the `ap_start` bit in the Control register must be set to 1. The HLS engine will then proceed and read any inputs grouped into the AXI4-Lite slave interface from the register in the interface.

When the block completes the operation, the `ap_done`, `ap_idle` and `ap_ready` registers will be set by the hardware output ports and the results for any output ports grouped into the `s_axilite` interface read from the appropriate register.

For function arguments, Vitis HLS automatically assigns the address for each argument or port that is assigned to the `s_axilite` interface. The tool will assign each port an offset starting from `0x10`, the lower addresses being reserved for control signals. The size, or range of addresses assigned to a port is dependent on the argument data type and the port protocol used.

Because the variables grouped into an AXI4-Lite interface are function arguments which do not have a default value in the C code, none of the argument registers in the `s_axilite` interface can be assigned a default value. The registers can be implemented with a reset using the `config_rtl` command, but they cannot be assigned any other default value.

The Control Register Map generated by Vitis HLS for the `ap_ctrl_hs` block control protocol is provided below:

```
-----Address Info-----
// 0x00 : Control signals
//      bit 0 - ap_start (Read/Write/COH)
//      bit 1 - ap_done (Read/COR)
//      bit 2 - ap_idle (Read)
//      bit 3 - ap_ready (Read)
```

```

//      bit 7 - auto_restart (Read/Write)
//      others - reserved
// 0x04 : Global Interrupt Enable Register
//      bit 0 - Global Interrupt Enable (Read/Write)
//      others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//      bit 0 - enable ap_done interrupt (Read/Write)
//      bit 1 - enable ap_ready interrupt (Read/Write)
//      others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//      bit 0 - ap_done (COR/TOW)
//      bit 1 - ap_ready (COR/TOW)
//      others - reserved
// 0x10 : Data signal of a
//      bit 7~0 - a[7:0] (Read/Write)
//      others - reserved
// 0x14 : reserved
// 0x18 : Data signal of b
//      bit 7~0 - b[7:0] (Read/Write)
//      others - reserved
//      : Control signal of b
//      bit 0 - b_ap_vld (Read/Write/SC)
//      others - reserved
// 0x20 : Data signal of c_i
//      bit 7~0 - c_i[7:0] (Read/Write)
//      others - reserved
// 0x24 : reserved
// 0x28 : Data signal of c_o
//      bit 7~0 - c_o[7:0] (Read)
//      others - reserved
// 0x2c : Control signal of c_o
//      bit 0 - c_o_ap_vld (Read/COR)
//      others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
// Clear on Handshake)

```

S_AXILITE and Port-Level Protocols

Port-level I/O protocols sequence data into and out of the HLS engine from the `s_axilite` adapter as seen in [S_AXILITE Example](#). In the Vivado IP flow, you can assign port-level I/O protocols to the individual ports and signals bundled into an `s_axilite` interface. In the Vitis kernel flow, changing the default port-level I/O protocols is not recommended unless necessary. The tool assigns a default port protocol to a port depending on the type and direction of the argument associated with it. The port can contain one or more of the following:

- Data signal for the argument
- Valid signal (`ap_vld/ap_ovld`) to indicate when the data can be read
- Acknowledge signal (`ap_ack`) to indicate when the data has been read

The default port protocol assignments for various argument types are as follows:

Argument Type	Default	Supported
scalar	<code>ap_none</code>	<code>ap_ack</code> and <code>ap_vld</code> can also be used
Pointers/References		

Argument Type	Default	Supported
Inputs	ap_none	ap_ack and ap_vld
Outputs	ap_vld	ap_none, ap_ack, and ap_ovld can also be used
Inouts	ap_ovld	ap_none, ap_ack, and ap_vld are also supported



IMPORTANT! Arrays default to `ap_memory`. The `bram` port protocol is not supported for arrays in an `s_axilite` interface.

The [S_AXILITE Example](#) groups port `b` into the `s_axilite` interface and specifies port `b` as using the `ap_vld` protocol with INTERFACE pragmas. As a result, the `s_axilite` adapter contains a register for the port `b` data, and a register for the port `b` input valid signal.

If the input valid register is not set to logic 1, the data in the `b` data register is not considered valid, and the design stalls and waits for the valid register to be set. Each time port `b` is read, Vitis HLS automatically clears the input valid register and resets the register to logic 0.



RECOMMENDED: To simplify the operation of your design, Xilinx recommends that you use the default port protocols associated with the `s_axilite` interface.

S_AXILITE Bundle Rules

In the [S_AXILITE Example](#) all the function arguments are grouped into a single `s_axilite` interface adapter specified by the `bundle=BUS_A` option in the INTERFACE pragma. The `bundle` option simply lets you group ports together into one interface.

In the Vitis kernel flow there should only be a single interface bundle, commonly named `s_axi_control` by the tool. So you should not specify the `bundle` option in that flow, or you will probably encounter an error during synthesis. However, in the Vivado IP flow you can specify multiple bundles using the `s_axilite` interface, and this will create a separate interface adapter for each bundle you have defined. The following example shows this:

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE mode=s_axilite port=a bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=b bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=c bundle=OUT
#pragma HLS INTERFACE mode=s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE mode=ap_vld port=b
    *c += *a + *b;
}
```

After synthesis completes, the Synthesis Summary report provides feedback regarding the number of `s_axilite` adapters generated. The SW-to-HW Mapping section of the report contains the HW info showing the control register offset and the address range for each port.

However, there are some rules related to using bundles with the `s_axilite` interface.

1. Default Bundle Names: This rule explicitly groups all interface ports with no bundle name into the same AXI4-Lite interface port, uses the tool default bundle name, and names the RTL port `s_axi_<default>`, typically `s_axi_control`.

In this example all ports are mapped to the default bundle:

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

2. User-Specified Bundle Names: This rule explicitly groups all interface ports with the same `bundle` name into the same AXI4-Lite interface port, and names the RTL port the value specified by `s_axi_<string>`.

The following example results in interfaces named `s_axi_BUS_A`, `s_axi_BUS_B`, and `s_axi_OUT`:

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE mode=s_axilite port=a bundle=BUS_A
#pragma HLS INTERFACE mode=s_axilite port=b bundle=BUS_B
#pragma HLS INTERFACE mode=s_axilite port=c bundle=OUT
#pragma HLS INTERFACE mode=s_axilite port=return bundle=OUT
#pragma HLS INTERFACE mode=ap_vld port=b
    *c += *a + *b;
}
```

3. Partially Specified Bundle Names: If you specify `bundle` names for some arguments, but leave other arguments unassigned, then the tool will bundle the arguments as follows:

- Group all ports into the specified bundles as indicated by the INTERFACE pragmas.
- Group any ports without bundle assignments into a default named bundle. The default name can either be the standard tool default, or an alternative default name if the tool default has already been specified by the user.

In the following example the user has specified `bundle=control`, which is the tool default name. In this case, port c will be assigned to `s_axi_control` as specified by the user, and the remaining ports will be bundled under `s_axi_control_r`, which is an alternative default name used by the tool.

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

S_AXILITE Offset Option

Note: The Vitis kernel flow determines the required offsets. Do not specify the `offset` option in that flow.

In the Vivado IP flow, Vitis HLS defines the size, or range of addresses assigned to a port in the [S_AXILITE Control Register Map](#) depending on the argument data type and the port protocol used. However, the INTERFACE pragma also contains an `offset` option that lets you specify the address offset in the AXI4-Lite interface.

When specifying the offset for your argument, you must consider the size of your data and reserve some extra for the port control protocol. The range of addresses you reserve should be based on a 32-bit word. You should reserve enough 32-bit words to fit your argument data type, and add one additional word for the control protocol, even for `ap_none`.



TIP: In the case of the `ap_memory` protocol for arrays, you do not need to reserve the extra word for the control protocol. In this case, simply reserve enough 32-bit words to fit your argument data type.

For example, to reserve enough space for a double you need to reserve 2 32-bit words for the 64-bit data type, and then reserve an additional 32-bit word for the control protocol. So you need to reserve a total of 3 32-bit words, or 96 bits. If your argument offset starts at `0x020`, then the next available offset would begin at `0x02c`, in order to reserve the required address range for your argument.

If you make a mistake in setting the offset of your arguments, by not reserving enough address range to fit your data type and the control protocol, Vitis HLS will recognize the error, will warn you of the issue, and will recover by moving your misplaced argument register to the end of the Control Register Map. This will allow your build to proceed, but may not work with your host application or driver if they were written to your specified offset.

C Driver Files

When an AXI4-Lite slave interface is implemented, a set of C driver files are automatically created. These C driver files provide a set of APIs that can be integrated into any software running on a CPU and used to communicate with the device via the AXI4-Lite slave interface.

The C driver files are created when the design is packaged as IP in the IP catalog.

Driver files are created for standalone and Linux modes. In standalone mode the drivers are used in the same way as any other Xilinx standalone drivers. In Linux mode, copy all the C files (`.c`) and header files (`.h`) files into the software project.

The driver files and API functions derive their name from the top-level function for synthesis. In the above example, the top-level function is called "example". If the top-level function was named "DUT" the name "example" would be replaced by "DUT" in the following description. The driver files are created in the packaged IP (located in the `impl` directory inside the `solution`).

Table 18: C Driver Files for a Design Named `example`

File Path	Usage Mode	Description
<code>data/example.mdd</code>	Standalone	Driver definition file.

Table 18: C Driver Files for a Design Named `example` (cont'd)

File Path	Usage Mode	Description
<code>data/example.tcl</code>	Standalone	Used by SDK to integrate the software into an SDK project.
<code>src/xexample_hw.h</code>	Both	Defines address offsets for all internal registers.
<code>src/xexample.h</code>	Both	API definitions
<code>src/xexample.c</code>	Both	Standard API implementations
<code>src/xexample_sinit.c</code>	Standalone	Initialization API implementations
<code>src/xexample_linux.c</code>	Linux	Initialization API implementations
<code>src/Makefile</code>	Standalone	Makefile

In file `xexample.h`, two structs are defined.

- **XExample_Config:** This is used to hold the configuration information (base address of each AXI4-Lite slave interface) of the IP instance.
- **XExample:** This is used to hold the IP instance pointer. Most APIs take this instance pointer as the first argument.

The standard API implementations are provided in files `xexample.c`, `xexample_sinit.c`, `xexample_linux.c`, and provide functions to perform the following operations.

- Initialize the device
- Control the device and query its status
- Read/write to the registers
- Set up, monitor, and control the interrupts

Refer to [Section IV: Vitis HLS C Driver Reference](#) for a description of the API functions provided in the C driver files.



IMPORTANT! The C driver APIs always use an unsigned 32-bit type (U32). You might be required to cast the data in the C code into the expected type.

C Driver Files and Float Types

C driver files always use a data 32-bit unsigned integer (U32) for data transfers. In the following example, the function uses float type arguments `a` and `r1`. It sets the value of `a` and returns the value of `r1`:

```
float caculate(float a, float *r1)
{
#pragma HLS INTERFACE mode=ap_vld register port=r1
#pragma HLS INTERFACE mode=s_axilite port=a
#pragma HLS INTERFACE mode=s_axilite port=r1
```

```
#pragma HLS INTERFACE mode=s_axilite port=return
*r1 = 0.5f*a;
return (a>0);
}
```

After synthesis, Vitis HLS groups all ports into the default AXI4-Lite interface and creates C driver files. However, as shown in the following example, the driver files use type U32:

```
// API to set the value of A
void XCaculate_SetA(XCaculate *InstancePtr, u32 Data) {
    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    XCaculate_WriteReg(InstancePtr->Hls_periph_bus_BaseAddress,
XCACULATE_HLS_PERIPH_BUS_ADDR_A_DATA, Data);
}

// API to get the value of R1
u32 XCaculate_GetR1(XCaculate *InstancePtr) {
    u32 Data;

    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

    Data = XCaculate_ReadReg(InstancePtr->Hls_periph_bus_BaseAddress,
XCACULATE_HLS_PERIPH_BUS_ADDR_R1_DATA);
    return Data;
}
```

If these functions work directly with float types, the write and read values are not consistent with expected float type. When using these functions in software, you can use the following casts in the code:

```
float a=3.0f,r1;
u32 ua,url1;

// cast float "a" to type U32
XCaculate_SetA(&calculate,*((u32*)&a));
url1=XCaculate_GetR1(&calculate);

// cast return type U32 to float type for "r1"
r1=*((float*)&url1);
```

Controlling Hardware



TIP: The example provided below demonstrates the *ap_ctrl_hs* block control protocol, which is the default for the Vivado IP flow. Refer to [Block-Level Control Protocols](#) for more information and a description of the *ap_ctrl_chain* protocol which is the default for the Vitis kernel flow.

In this example, the hardware header file `xexample_hw.h` provides a complete list of the memory mapped locations for the ports grouped into the AXI4-Lite slave interface, as described in [S_AXILITE Control Register Map](#).

```
// 0x00 : Control signals
//      bit 0 - ap_start (Read/Write/SC)
//      bit 1 - ap_done (Read/COR)
//      bit 2 - ap_idle (Read)
//      bit 3 - ap_ready (Read)
//      bit 7 - auto_restart (Read/Write)
//      others - reserved
// 0x04 : Global Interrupt Enable Register
//      bit 0 - Global Interrupt Enable (Read/Write)
//      others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//      bit 0 - Channel 0 (ap_done)
//      bit 1 - Channel 1 (ap_ready)
// 0x0c : IP Interrupt Status Register (Read/TOW)
//      bit 0 - Channel 0 (ap_done)
//      others - reserved
// 0x10 : Data signal of a
//      bit 7~0 - a[7:0] (Read/Write)
//      others - reserved
// 0x14 : reserved
// 0x18 : Data signal of b
//      bit 7~0 - b[7:0] (Read/Write)
//      others - reserved
// 0x1c : reserved
// 0x20 : Data signal of c_i
//      bit 7~0 - c_i[7:0] (Read/Write)
//      others - reserved
// 0x24 : reserved
// 0x28 : Data signal of c_o
//      bit 7~0 - c_o[7:0] (Read)
//      others - reserved
// 0x2c : Control signal of c_o
//      bit 0 - c_o_ap_vld (Read/COR)
//      others - reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH =
Clear on
Handshake)
```

To correctly program the registers in the `s_axilite` interface, you must understand how the hardware ports operate with the default port protocols, or the custom protocols as described in [S_AXILITE and Port-Level Protocols](#).

For example, to start the block operation the `ap_start` register must be set to 1. The device will then proceed and read any inputs grouped into the AXI4-Lite slave interface from the register in the interface. When the block completes operation, the `ap_done`, `ap_idle` and `ap_ready` registers will be set by the hardware output ports and the results for any output ports grouped into the AXI4-Lite slave interface read from the appropriate register.

The implementation of function argument `c` in the example highlights the importance of some understanding how the hardware ports operate. Function argument `c` is both read and written to, and is therefore implemented as separate input and output ports `c_i` and `c_o`, as explained in [S_AXILITE Example](#).

The first recommended flow for programming the `s_axilite` interface is for a one-time execution of the function:

- Use the interrupt function standard API implementations provided in the [C Driver Files](#) to determine how you want the interrupt to operate.
- Load the register values for the block input ports. In the above example this is performed using API functions `XExample_Set_a`, `XExample_Set_b`, and `XExample_Set_c_i`.
- Set the `ap_start` bit to 1 using `XExample_Start` to start executing the function. This register is self-clearing as noted in the header file above. After one transaction, the block will suspend operation.
- Allow the function to execute. Address any interrupts which are generated.
- Read the output registers. In the above example this is performed using API functions `XExample_Get_c_o_vld`, to confirm the data is valid, and `XExample_Get_c_o`.

Note: The registers in the `s_axilite` interface obey the same I/O protocol as the ports. In this case, the output valid is set to logic 1 to indicate if the data is valid.

- Repeat for the next transaction.

The second recommended flow is for continuous execution of the block. In this mode, the input ports included in the AXI4-Lite interface should only be ports which perform configuration. The block will typically run much faster than a CPU. If the block must wait for inputs, the block will spend most of its time waiting:

- Use the interrupt function to determine how you wish the interrupt to operate.
- Load the register values for the block input ports. In the above example this is performed using API functions `XExample_Set_a`, `XExample_Set_a` and `XExample_Set_c_i`.
- Set the auto-start function using API `XExample_EnableAutoRestart`.
- Allow the function to execute. The individual port I/O protocols will synchronize the data being processed through the block.
- Address any interrupts which are generated. The output registers could be accessed during this operation but the data may change often.
- Use the API function `XExample_DisableAutoRestart` to prevent any more executions.
- Read the output registers. In the above example this is performed using API functions `XExample_Get_c_o` and `XExample_Set_c_o_vld`.

Controlling Software

The API functions can be used in the software running on the CPU to control the hardware block. An overview of the process is:

- Create an instance of the hardware
- Look Up the device configuration

- Initialize the device
- Set the input parameters of the HLS block
- Start the device and read the results

An example application is shown below.

```
#include "xexample.h"      // Device driver for HLS HW block
#include "xparameters.h"

// HLS HW instance
XExample HlsExample;
XExample_Config *ExamplePtr

int main() {
    int res_hw;

    // Look Up the device configuration
    ExamplePtr = XExample_LookupConfig(XPAR_XEXAMPLE_0_DEVICE_ID);
    if (!ExamplePtr) {
        print("ERROR: Lookup of accelerator configuration failed.\n\r");
        return XST_FAILURE;
    }

    // Initialize the Device
    status = XExample_CfgInitialize(&HlsExample, ExamplePtr);
    if (status != XST_SUCCESS) {
        print("ERROR: Could not initialize accelerator.\n\r");
        exit(-1);
    }

    //Set the input parameters of the HLS block
    XExample_Set_a(&HlsExample, 42);
    XExample_Set_b(&HlsExample, 12);
    XExample_Set_c_i(&HlsExample, 1);

    // Start the device and read the results
    XExample_Start(&HlsExample);
    do {
        res_hw = XExample_Get_c_o(&HlsExample);
    } while (XExample_Get_c_o(&HlsExample) == 0); // wait for valid data output
    print("Detected HLS peripheral complete. Result received.\n\r");
}
```

Control Clock and Reset in AXI4-Lite Interfaces

Note: If you instantiate the slave AXI4-Lite register file in a bus fabric that uses a different clock frequency, Vivado IP integrator will automatically generate a clock domain crossing (CDC) slice that performs the same function as the control clock described below, making use of the option unnecessary.

By default, Vitis HLS uses the same clock for the AXI4-Lite interface and the synthesized design. Vitis HLS connects all registers in the AXI4-Lite interface to the clock used for the synthesized logic (`ap_clk`).

Optionally, you can use the INTERFACE directive `clock` option to specify a separate clock for each AXI4-Lite port. When connecting the clock to the AXI4-Lite interface, you must use the following protocols:

- AXI4-Lite interface clock must be synchronous to the clock used for the synthesized logic (`ap_clk`). That is, both clocks must be derived from the same master generator clock.
- AXI4-Lite interface clock frequency must be equal to or less than the frequency of the clock used for the synthesized logic (`ap_clk`).

If you use the `clock` option with the INTERFACE directive, you only need to specify the `clock` option on one function argument in each bundle. Vitis HLS implements all other function arguments in the bundle with the same clock and reset. Vitis HLS names the generated reset signal with the prefix `ap_rst_` followed by the clock name. The generated reset signal is active-Low independent of the `config_rtl` command.

The following example shows how Vitis HLS groups function arguments `a` and `b` into an AXI4-Lite port with a clock named `AXI_clk1` and an associated reset port.

```
// Default AXI-Lite interface implemented with independent clock called
// AXI_clk1
#pragma HLS interface mode=s_axilite port=a clock=AXI_clk1
#pragma HLS interface mode=s_axilite port=b
```

In the following example, Vitis HLS groups function arguments `c` and `d` into AXI4-Lite port `CTRL1` with a separate clock called `AXI_clk2` and an associated reset port.

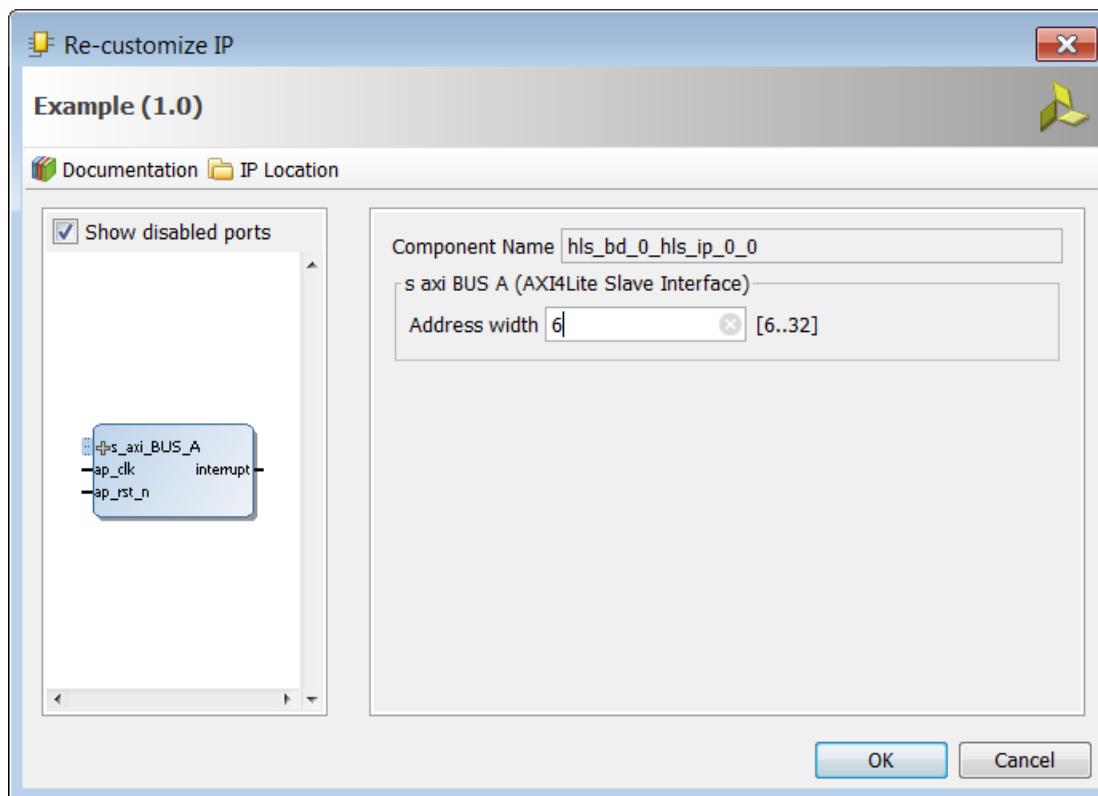
```
// CTRL1 AXI-Lite bundle implemented with a separate clock (called AXI_clk2)
#pragma HLS interface mode=s_axilite port=c bundle=CTRL1 clock=AXI_clk2
#pragma HLS interface mode=s_axilite port=d bundle=CTRL1
```

Customizing AXI4-Lite Slave Interfaces in IP Integrator

When an HLS RTL design using an AXI4-Lite slave interface is incorporated into a design in Vivado IP integrator, you can customize the block. From the block diagram in IP integrator, select the HLS block, right-click with the mouse button and select **Customize Block**.

The address width is by default configured to the minimum required size. Modify this to connect to blocks with address sizes less than 32-bit.

Figure 71: Customizing AXI4-Lite Slave Interfaces in IP Integrator



AXI4-Stream Interfaces

An AXI4-Stream interface can be applied to any input argument and any array or pointer output argument. Because an AXI4-Stream interface transfers data in a sequential streaming manner, it cannot be used with arguments that are both read and written. In terms of data layout, the data type of the AXI4-Stream is aligned to the next byte. For example, if the size of the data type is 12 bits, it will be extended to 16 bits. Depending on whether a signed/unsigned interface is selected, the extended bits are either sign-extended or zero-extended.

If the stream data type is an user-defined struct, the default procedure is to keep the struct aggregated and align the struct to the size of the largest data element to the nearest byte. The only exception to this rule is if the struct contains a `hls::stream` object. In this special case, the struct will be disaggregated and an axi stream will be created for each member element of the struct.



TIP: The maximum supported port width is 4096 bits, even for aggregated structs or reshaped arrays.

The following code examples show how the packed alignment depends on your struct type. If the struct contains only char type, as shown in the following example, then it will be packed with alignment of one byte. Total size of the struct will be two bytes:

```
struct A {
    char foo;
    char bar;
};
```

However, if the struct has elements with different data types, as shown below, then it will be packed and aligned to the size of the largest data element, or four bytes in this example. Element bar will be padded with three bytes resulting in a total size of eight bytes for the struct:

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

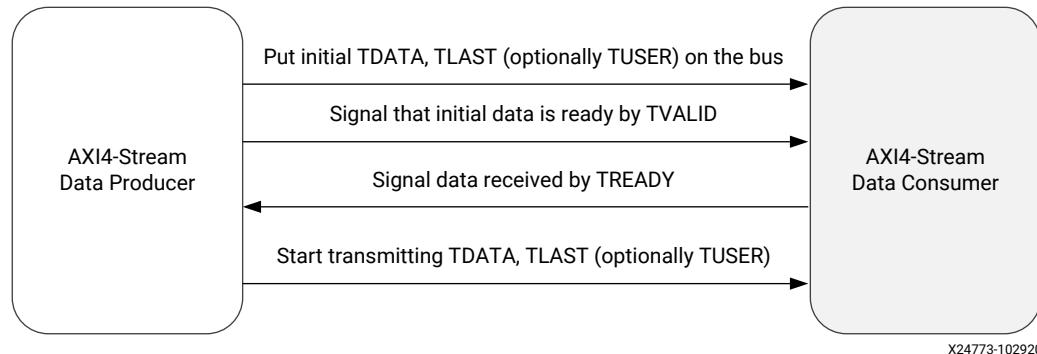


IMPORTANT! Structs contained in AXI4-Stream interfaces (axis) are aggregated by default, and the stream itself cannot be disaggregated. If separate streams for member elements of the struct are desired then this must be manually coded as separate elements, resulting in a separate axis interface for each element.

How AXI4-Stream Works

AXI4-Stream is a protocol designed for transporting arbitrary unidirectional data. In an AXI4-Stream, TDATA width of bits is transferred per clock cycle. The transfer is started once the producer sends the TVALID signal and the consumer responds by sending the TREADY signal (once it has consumed the initial TDATA). At this point, the producer will start sending TDATA and TLAST (TUSER if needed to carry additional user-defined sideband data). TLAST signals the last byte of the stream. So the consumer keeps consuming the incoming TDATA until TLAST is asserted.

Figure 72: AXI4-Stream Handshake



AXI4-Stream has additional optional features like sending positional data with TKEEP and TSTRB ports which makes it possible to multiplex both the data position and data itself on the TDATA signal. Using the TID and TDIST signals, you can route streams as these fields roughly corresponds to stream identifier and stream destination identifier. Refer to *Vivado Design Suite: AXI Reference Guide (UG1037)* or the *AMBA AXI4-Stream Protocol Specification (ARM IHI 0051A)* for more information.

How AXI4-Stream is Implemented

If your design requires a streaming interface, lets first start with defining and using a streaming data structure like `hls::stream` in Vitis HLS. This simple object encapsulates the requirements of streaming and its streaming interface is by default implemented in the RTL as a FIFO interface (`ap_fifo`) but can be optionally, implemented as a handshake interface (`ap_hs`) or an AXI4-Stream interface (`axis`).

If a AXI4-Stream interface (`axis`) is specified via the interface pragma mode option, the interface implementation will mimic the style of an AXIS interface by defining the TDATA, TVALID and TREADY signals.

If a more formal AXIS implementation is desired, then Vitis HLS requires the usage of a special data type (`hls::axis` defined in `ap_axi_sdata.h`) to encapsulate the requirements of the AXI4-Stream protocol and implement the special RTL signals needed for this interface.

The AXI4-Stream interface is implemented as a struct type in Vitis HLS and has the following signature (defined in `ap_axi_sdata.h`):

```
template <typename T, size_t WUser, size_t WId, size_t WDest> struct axis
{ .. };
```

Where:

- **T:** The data type to be streamed.



TIP: This can support any data type, including `ap_fixed`.

- **WUser:** Width of the TUSER signal
- **WId:** Width of the TID signal
- **WDest:** Width of the TDest signal

When the stream data type (`T`) are simple integer types, there are two predefined types of AXI4-Stream implementations available:

- A signed implementation of the AXI4-Stream class (or more simply `ap_axis<Wdata, WUser, WId, WDest>`)

```
hls::axis<ap_int<WData>, WUser, WId, WDest>
```

- An unsigned implementation of the AXI4-Stream class (or more simply `ap_axiu<WData, WUser, WId, WDest>`)

```
hls::axis<ap_uint<WData>, WUser, WId, WDest>
```

The value specified for the `WUser`, `WId`, and `WDest` template parameters controls the usage of side-channel signals in the AXI4-Stream interface.

When the `hls::axis` class is used, the generated RTL will typically contain the actual data signal `TDATA`, and the following additional signals: `TVALID`, `TREADY`, `TKEEP`, `TSTRB`, `TLAST`, `TUSER`, `TID`, and `TDEST`.

`TVALID`, `TREADY`, and `TLAST` are necessary control signals for the AXI4-Stream protocol. `TKEEP`, `TSTRB`, `TUSER`, `TID`, and `TDEST` signals are optional special signals that can be used to pass around additional bookkeeping data.



TIP: If `WUser`, `WId`, and `WDest` are set to 0, the generated RTL will not include the optional `TUSER`, `TID`, and `TDEST` signals in the interface.

Registered AXI4-Stream Interfaces

As a default, AXI4-Stream interfaces are always implemented as registered interfaces to ensure that no combinational feedback paths are created when multiple HLS IP blocks with AXI4-Stream interfaces are integrated into a larger design. For AXI4-Stream interfaces, four types of register modes are provided to control how the interface registers are implemented:

- **Forward:** Only the `TDATA` and `TVALID` signals are registered.
- **Reverse:** Only the `TREADY` signal is registered.
- **Both:** All signals (`TDATA`, `TREADY`, and `TVALID`) are registered. This is the default.
- **Off:** None of the port signals are registered.

The AXI4-Stream side-channel signals are considered to be data signals and are registered whenever `TDATA` is registered.



RECOMMENDED: When connecting HLS generated IP blocks with AXI4-Stream interfaces at least one interface should be implemented as a registered interface or the blocks should be connected via an AXI4-Stream Register Slice.

There are two basic methods to use an AXI4-Stream in your design:

- Use an AXI4-Stream without side-channels.
- Use an AXI4-Stream with side-channels.

This second use model provides additional functionality, allowing the optional side-channels which are part of the AXI4-Stream standard, to be used directly in your C/C++ code.

AXI4-Stream Interfaces without Side-Channels

An AXI4-Stream is used without side-channels when the function argument, `ap_axis` or `ap_axiu` data type, does not contain any AXI4 side-channel elements (that is, when the `WUser`, `WId`, and `WDest` parameters are set to 0). In the following example, both interfaces are implemented using an AXI4-Stream:

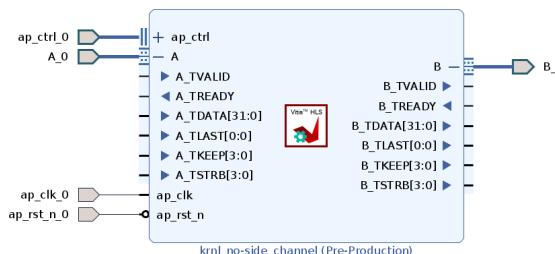
```
#include "ap_axi_sdata.h"
#include "hls_stream.h"

typedef ap_axiu<32, 0, 0, 0> trans_pkt;

void example(hls::stream< trans_pkt > &A, hls::stream< trans_pkt > &B)
{
#pragma HLS INTERFACE mode=axis port=A
#pragma HLS INTERFACE mode=axis port=B
    trans_pkt tmp;
    A.read(tmp);
    tmp.data += 5;
    B.write(tmp);
}
```

After synthesis, both arguments are implemented with a data port (`TDATA`) and the standard AXI4-Stream protocol ports, `TVALID`, `TREADY`, `TKEEP`, `TLAST`, and `TSTRB`, as shown in the following figure.

Figure 73: AXI4-Stream Interfaces without Side-Channels



TIP: If you specify an `hls::stream` object with a data type other than `ap_axis` or `ap_axiu`, the tool will infer an AXI4-Stream interface without the `TLAST` signal, or any of the side-channel signals. This implementation of the AXI4-Stream interface consumes fewer device resources, but offers no visibility into when the stream is ending.

Multiple variables can be combined into the same AXI4-Stream interface by using a struct, which is aggregated by Vitis HLS by default. Aggregating the elements of a struct into a single wide-vector, allows all elements of the struct to be implemented in the same AXI4-Stream interface.

AXI4-Stream Interfaces with Side-Channels

The following example shows how the side-channels can be used directly in the C/C++ code and implemented on the interface. The code uses `#include "ap_axi_sdata.h"` to provide an API to handle the side-channels of the AXI4-Stream interface. In the following example a signed 32-bit data type is used:

```

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

#define DWIDTH 32

typedef ap_axiu<DWIDTH, 1, 1, 1> trans_pkt;

extern "C" {
    void krnl_stream_vmult(hls::stream<trans_pkt> &A,
                           hls::stream<trans_pkt> &B) {
#pragma HLS INTERFACE mode=axis port=A
#pragma HLS INTERFACE mode=axis port=B
#pragma HLS INTERFACE mode=s_axilite port=return bundle=control
    bool eos = false;

    vmult: do {
#pragma HLS PIPELINE II=1
        trans_pkt t2 = A.read();

        // Packet for Output
        trans_pkt t_out;

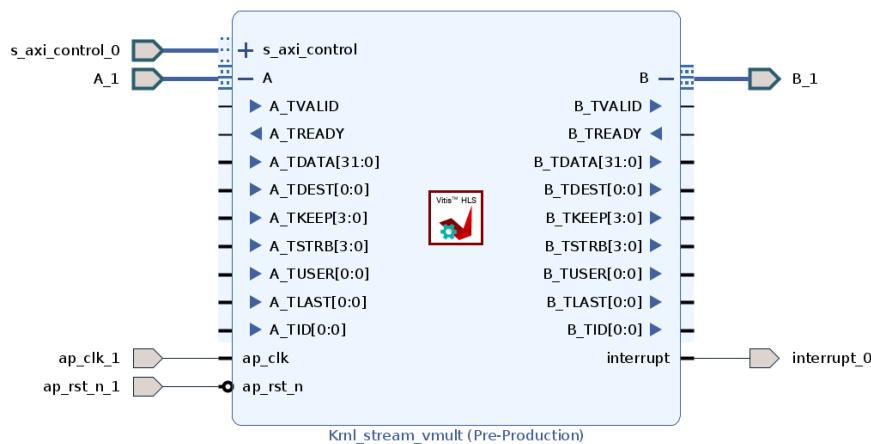
        // Reading data from input packet
        ap_uint<DWIDTH> in2 = t2.data;
        ap_uint<DWIDTH> tmpOut = in2 * 5;

        // Setting data and configuration to output packet
        t_out.data = tmpOut;
        t_out.last = t2.last;
        t_out.keep = -1; //Enabling all bytes
        // Writing packet to output stream
        B.write(t_out);
        if (t2.last) {
            eos = true;
        }
    } while (eos == false);
}
}

```

After synthesis, both the A and B arguments are implemented with data ports, the standard AXI4-Stream protocol ports, TVALID and TREADY and all of the optional ports described in the struct.

Figure 74: AXI4-Stream Interfaces with Side-Channels



Coding Style for Array to Stream

You should perform all the operations on temp variables. Read the input stream, process the temp variable, and write the output stream, as shown in the example below. This approach lets you preserve the sequential reading and writing of the stream of data, rather than attempting multiple or random reads or writes.

```

struct A {
    short varA;
    int varB;
};

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

```

If this coding style is not adhered to, it will lead to functional failures of the stream processing.

Port-Level I/O Protocols



IMPORTANT! The port-level I/O protocols of interfaces defined in the Vitis kernel flow are set by design and should not be modified as a general rule.

By default input pointers and pass-by-value arguments are implemented as simple wire ports with no associated handshaking signal. For example, in the `vadd` function discussed in [Interfaces for Vivado IP Flow](#), the input ports are implemented without an I/O protocol, only a data port. If the port has no I/O protocol, (by default or by design) the input data must be held stable until it is read.

By default output pointers are implemented with an associated output valid signal to indicate when the output data is valid. In the `vadd` function example, the output port is implemented with an associated output valid port (`out_r_o_ap_vld`) which indicates when the data on the port is valid and can be read. If there is no I/O protocol associated with the output port, it is difficult to know when to read the data.



TIP: It is always a good idea to use an I/O protocol on an output.

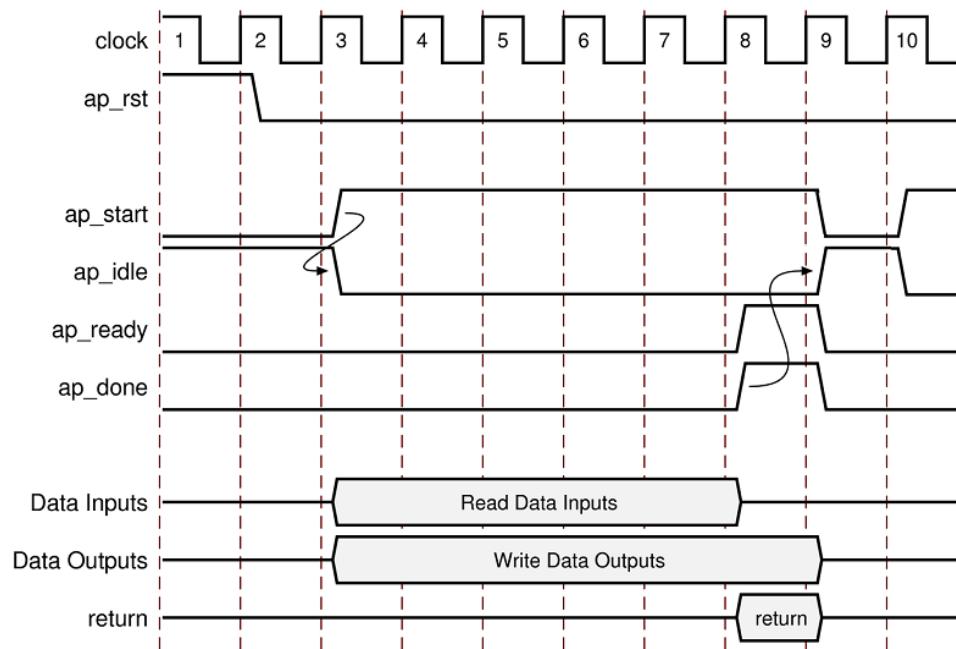
Function arguments which are both read from and written to are split into separate input and output ports. In the `vadd` function example, the `out_r` argument is implemented as both an input port `out_r_i`, and an output port `out_r_o` with associated I/O protocol port `out_r_o_ap_vld`.

If the function has a return value, an output port `ap_return` is implemented to provide the return value. When the RTL design completes one transaction, this is equivalent to one execution of the C/C++ function, the block-level protocols indicate the function is complete with the `ap_done` signal. This also indicates the data on port `ap_return` is valid and can be read.

Note: The return value of the top-level function cannot be a pointer.

For the example code shown the timing behavior is shown in the following figure (assuming that the target technology and clock frequency allow a single addition per clock cycle).

Figure 75: RTL Port Timing with Default Synthesis



- The design starts when `ap_start` is asserted High.
- The `ap_idle` signal is asserted Low to indicate the design is operating.
- The input data is read at any clock after the first cycle. Vitis HLS schedules when the reads occur. The `ap_ready` signal is asserted High when all inputs have been read.
- When output `sum` is calculated, the associated output handshake (`sum_o_ap_vld`) indicates that the data is valid.
- When the function completes, `ap_done` is asserted. This also indicates that the data on `ap_return` is valid.
- Port `ap_idle` is asserted High to indicate that the design is waiting start again.

Port-Level I/O: No Protocol

The `ap_none` specifies that no I/O protocol be added to the port. When this is specified the argument is implemented as a data port with no other associated signals. The `ap_none` mode is the default for scalar inputs.

ap_none

The `ap_none` port-level I/O protocol is the simplest interface type and has no other signals associated with it. Neither the input nor output data signals have associated control ports that indicate when data is read or written. The only ports in the RTL design are those specified in the source code.

An `ap_none` interface does not require additional hardware overhead. However, the `ap_none` interface does requires the following:

- Producer blocks to do one of the following:
 - Provide data to the input port at the correct time
 - Hold data for the length of a transaction until the design completes
- Consumer blocks to read output ports at the correct time

Note: The `ap_none` interface cannot be used with array arguments.

Port-Level I/O: Wire Handshakes

Interface mode `ap_hs` includes a two-way handshake signal with the data port. The handshake is an industry standard valid and acknowledge handshake. Mode `ap_vld` is the same but only has a valid port and `ap_ack` only has a acknowledge port.

Mode `ap_ovld` is for use with in-out arguments. When the in-out is split into separate input and output ports, mode `ap_none` is applied to the input port and `ap_vld` applied to the output port. This is the default for pointer arguments that are both read and written.

The `ap_hs` mode can be applied to arrays that are read or written in sequential order. If Vitis HLS can determine the read or write accesses are not sequential, it will halt synthesis with an error. If the access order cannot be determined, Vitis HLS will issue a warning.

ap_hs (ap_ack, ap_vld, and ap_ovld)

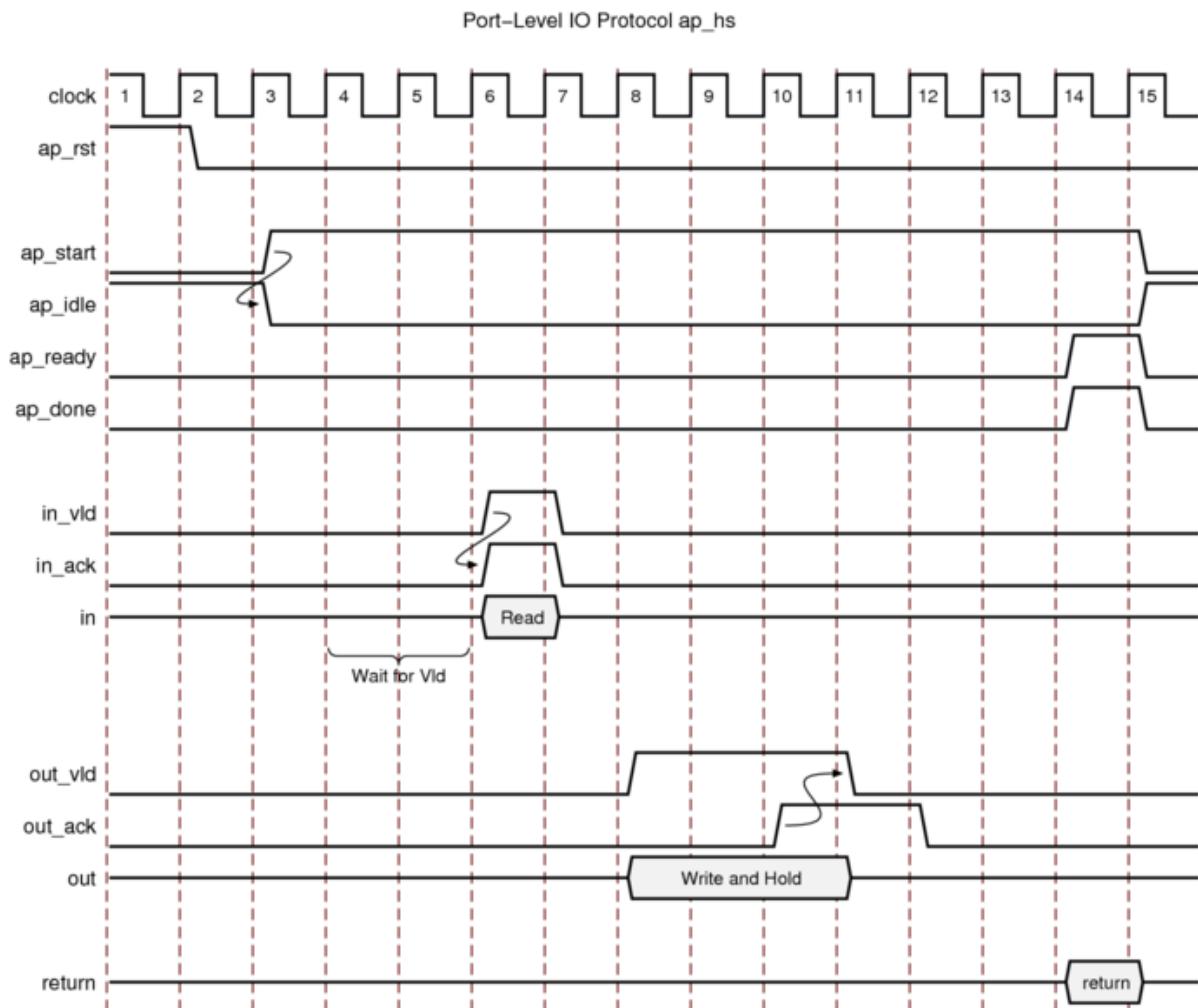
The `ap_hs` port-level I/O protocol provides the greatest flexibility in the development process, allowing both bottom-up and top-down design flows. Two-way handshakes safely perform all intra-block communication, and manual intervention or assumptions are not required for correct operation. The `ap_hs` port-level I/O protocol provides the following signals:

- Data port
- Valid signal to indicate when the data signal is valid and can be read
- Acknowledge signal to indicate when the data has been read

The following figure shows how an `ap_hs` interface behaves for both an input and output port. In this example, the input port is named `in`, and the output port is named `out`.

Note: The control signals names are based on the original port name. For example, the `valid` port for data input `in` is named `in_vld`.

Figure 76: Behavior of ap_hs Interface



For inputs, the following occurs:

- After start is applied, the block begins normal operation.
- If the design is ready for input data but the input `valid` is Low, the design stalls and waits for the input `valid` to be asserted to indicate a new input value is present.

Note: The preceding figure shows this behavior. In this example, the design is ready to read data input `in` on clock cycle 4 and stalls waiting for the input `valid` before reading the data.

- When the input `valid` is asserted High, an output acknowledge is asserted High to indicate the data was read.

For outputs, the following occurs:

- After start is applied, the block begins normal operation.
- When an output port is written to, its associated output `valid` signal is simultaneously asserted to indicate valid data is present on the port.
- If the associated input acknowledge is Low, the design stalls and waits for the input acknowledge to be asserted.
- When the input acknowledge is asserted, indicating the data has been read, the output `valid` is deasserted on the next clock edge.

ap_ack

The `ap_ack` port-level I/O protocol is a subset of the `ap_hs` interface type. The `ap_ack` port-level I/O protocol provides the following signals:

- Data port
- Acknowledge signal to indicate when data is consumed
 - For input arguments, the design generates an output acknowledge that is active-High in the cycle the input is read.
 - For output arguments, Vitis HLS implements an input acknowledge port to confirm the output was read.

Note: After a write operation, the design stalls and waits until the input acknowledge is asserted High, which indicates the output was read by a consumer block. However, there is no associated output port to indicate when the data can be consumed.



CAUTION! You cannot use C/RTL co-simulation to verify designs that use `ap_ack` on an output port.

ap_vld

The `ap_vld` is a subset of the `ap_hs` interface type. The `ap_vld` port-level I/O protocol provides the following signals:

- Data port
- Valid signal to indicate when the data signal is valid and can be read
 - For input arguments, the design reads the data port as soon as the `valid` is active. Even if the design is not ready to read new data, the design samples the data port and holds the data internally until needed.
 - For output arguments, Vitis HLS implements an output `valid` port to indicate when the data on the output port is valid.

ap_ovld

The `ap_ovld` is a subset of the `ap_hs` interface type. The `ap_ovld` port-level I/O protocol provides the following signals:

- Data port
- Valid signal to indicate when the data signal is valid and can be read
 - For input arguments and the input half of inout arguments, the design defaults to type `ap_none`.
 - For output arguments and the output half of inout arguments, the design implements type `ap_vld`.

Port-Level I/O: Memory Interface Protocol

Array arguments are implemented by default as an `ap_memory` interface. This is a standard block RAM interface with data, address, chip-enable, and write-enable ports.

An `ap_memory` interface can be implemented as a single-port or dual-port interface. If Vitis HLS can determine that using a dual-port interface will reduce the initial interval, it will automatically implement a dual-port interface. The `BIND_STORAGE` pragma or directive is used to specify the memory resource and if this directive is specified on the array with a single-port block RAM, a single-port interface will be implemented. Conversely, if a dual-port interface is specified using the `BIND_STORAGE` pragma and Vitis HLS determines this interface provides no benefit it will automatically implement a single-port interface.

If the array is accessed in a sequential manner an `ap_fifo` interface can be used. As with the `ap_hs` interface, Vitis HLS will halt if it determines the data access is not sequential, report a warning if it cannot determine if the access is sequential or issue no message if it determines the access is sequential. The `ap_fifo` interface can only be used for reading or writing, not both.

`ap_memory`, `bram`

The `ap_memory` and `bram` interface port-level I/O protocols are used to implement array arguments. This type of port-level I/O protocol can communicate with memory elements (for example, RAMs and ROMs) when the implementation requires random accesses to the memory address locations.

Note: If you only need sequential access to the memory element, use the `ap_fifo` interface instead. The `ap_fifo` interface reduces the hardware overhead, because address generation is not performed.

The `ap_memory` and `bram` interface port-level I/O protocols are identical. The only difference is the way Vivado IP integrator shows the blocks:

- The `ap_memory` interface appears as discrete ports.
- The `bram` interface appears as a single, grouped port. In IP integrator, you can use a single connection to create connections to all ports.

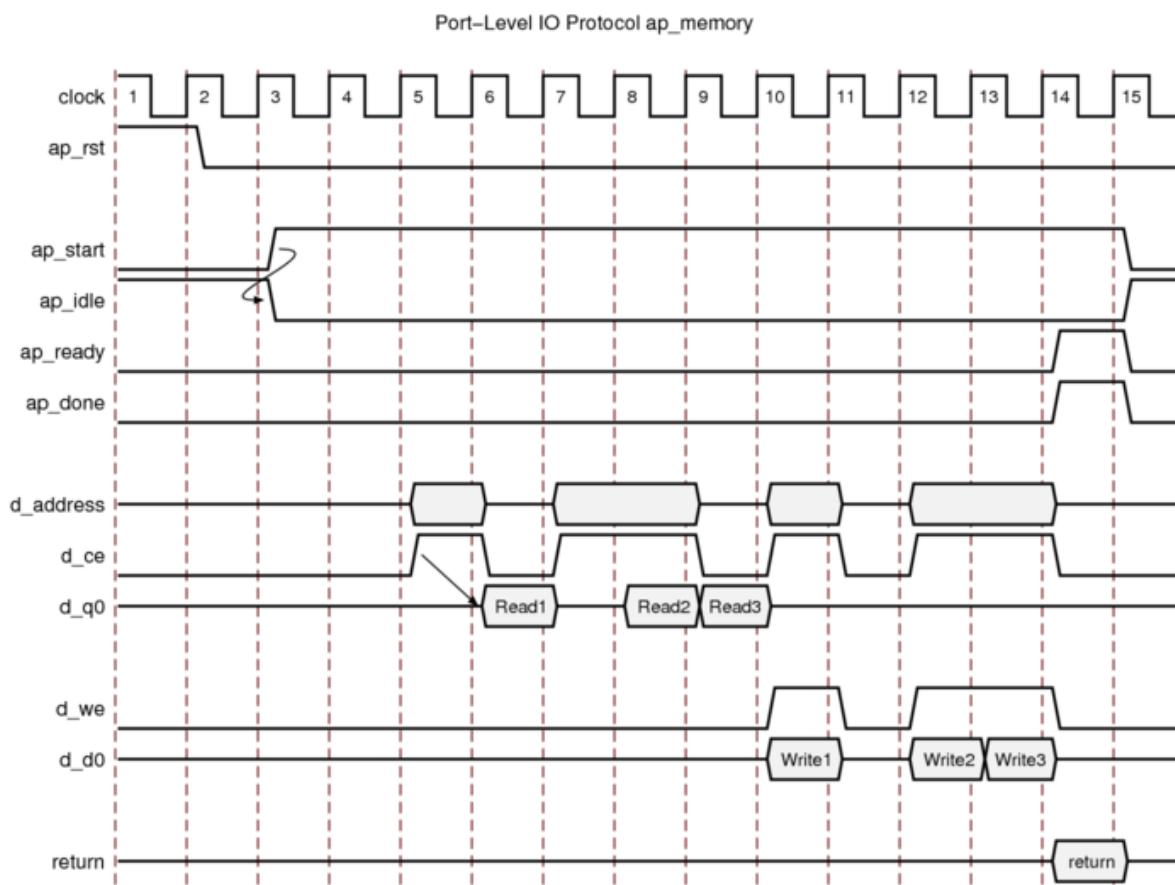
When using an `ap_memory` interface, specify the array targets using the `BIND_STORAGE` pragma. If no target is specified for the arrays, Vitis HLS determines whether to use a single or dual-port RAM interface.



TIP: Before running synthesis, ensure array arguments are targeted to the correct memory type using the `BIND_STORAGE` pragma. Re-synthesizing with corrected memories can result in a different schedule and RTL.

The following figure shows an array named `d` specified as a single-port block RAM. The port names are based on the C/C++ function argument. For example, if the C/C++ argument is `d`, the chip-enable is `d_ce`, and the input data is `d_d0` based on the `output/q` port of the BRAM.

Figure 77: Behavior of ap_memory Interface



After reset, the following occurs:

- After start is applied, the block begins normal operation.
- Reads are performed by applying an address on the output address ports while asserting the output signal `d_ce`.

Note: For a default block RAM, the design expects the input data `d_q0` to be available in the next clock cycle. You can use the `BIND_STORAGE` pragma to indicate the RAM has a longer read latency.

- Write operations are performed by asserting output ports `d_ce` and `d_we` while simultaneously applying the address and output data `d_d0`.

ap_fifo

When an output port is written to, its associated output `valid` signal interface is the most hardware-efficient approach when the design requires access to a memory element and the access is always performed in a sequential manner, that is, no random access is required. The `ap_fifo` port-level I/O protocol supports the following:

- Allows the port to be connected to a FIFO
- Enables complete, two-way `empty`-`full` communication
- Works for arrays, pointers, and pass-by-reference argument types

Note: Functions that can use an `ap_fifo` interface often use pointers and might access the same variable multiple times. To understand the importance of the `volatile` qualifier when using this coding style, see [Multi-Access Pointers on the Interface](#).

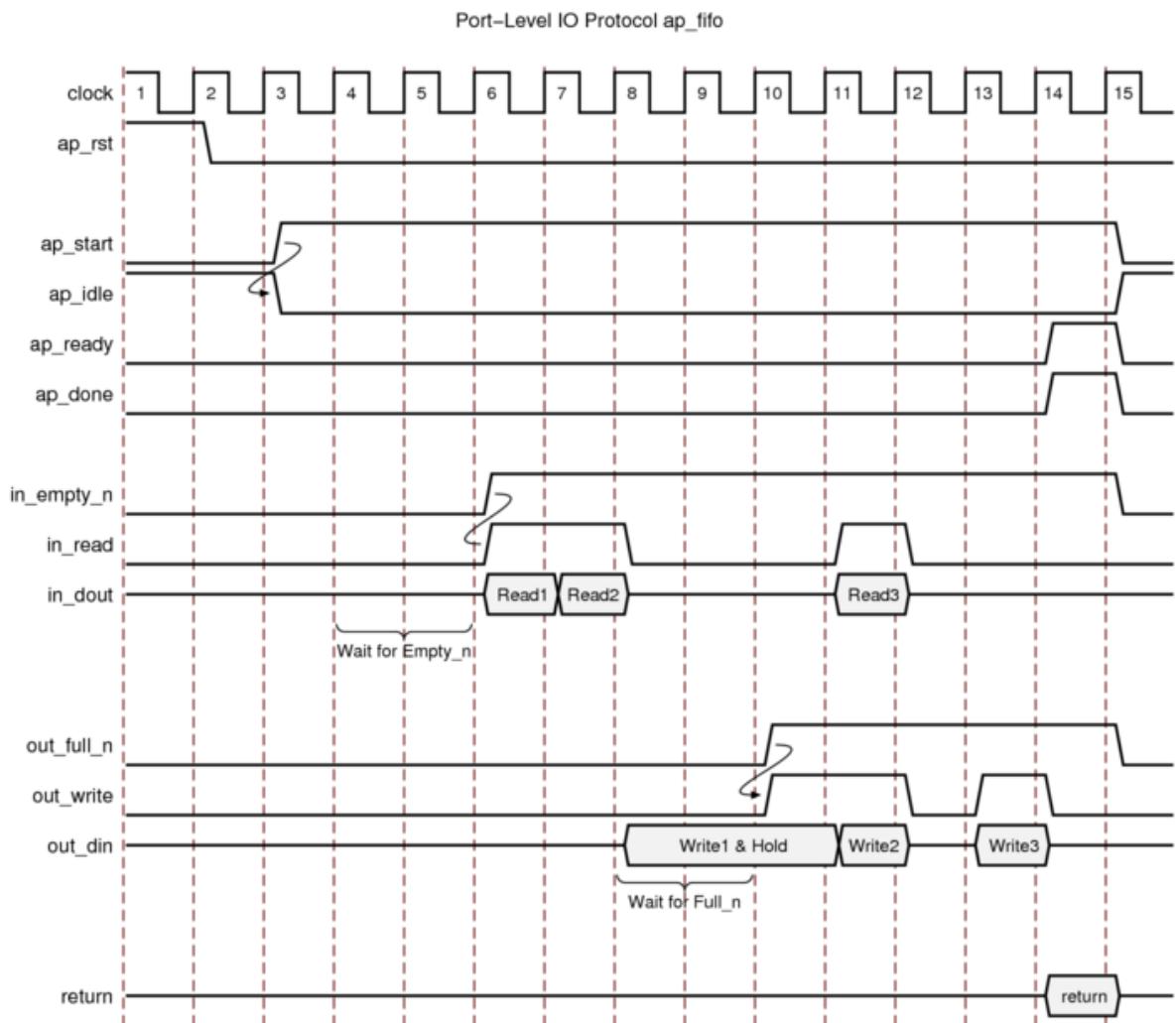
In the following example, `in1` is a pointer that accesses the current address, then two addresses above the current address, and finally one address below.

```
void foo(int* in1, ...) {
    int data1, data2, data3;
    ...
    data1= *in1;
    data2= *(in1+2);
    data3= *(in1-1);
    ...
}
```

If `in1` is specified as an `ap_fifo` interface, Vitis HLS checks the accesses, determines the accesses are not in sequential order, issues an error, and halts. To read from non-sequential address locations, use an `ap_memory` or `bram` interface.

You cannot specify an `ap_fifo` interface on an argument that is both read from and written to. You can only specify an `ap_fifo` interface on an input or an output argument. A design with input argument `in` and output argument `out` specified as `ap_fifo` interfaces behaves as shown in the following figure.

Figure 78: Behavior of ap_fifo Interface



For inputs, the following occurs:

- After ap_start is applied, the block begins normal operation.
- If the input port is ready to be read but the FIFO is empty as indicated by input port `in_empty_n` Low, the design stalls and waits for data to become available.
- When the FIFO contains data as indicated by input port `in_empty_n` High, an output acknowledge `in_read` is asserted High to indicate the data was read in this cycle.

For outputs, the following occurs:

- After start is applied, the block begins normal operation.
- If an output port is ready to be written to but the FIFO is full as indicated by `out_full_n` Low, the data is placed on the output port but the design stalls and waits for the space to become available in the FIFO.

- When space becomes available in the FIFO as indicated by `out_full_n` High, the output acknowledge signal `out_write` is asserted to indicate the output data is `valid`.
 - If the top-level function or the top-level loop is pipelined using the `-rewind` option, Vitis HLS creates an additional output port with the suffix `_lwr`. When the last write to the FIFO interface completes, the `_lwr` port goes active-High.
-

Block-Level Control Protocols

The execution mode of a Vitis kernel or Vivado IP is specified by the block-level control protocol. Execution modes of kernels include:

- Pipelined execution (`ap_ctrl_chain`) permitting overlapping kernel runs to begin processing additional data as soon as the kernel is ready.
- Sequential execution (`ap_ctrl_hs`) requiring the kernel to complete one cycle before beginning another.
- Data driven execution (`ap_ctrl_none`) which enables the kernel to run when data is available, and stall when data is not.

You can specify the block-level control protocol on the function or the function return. If the C/C++ code does not return a value, you can still specify the control protocol on the function return. If the C/C++ code uses a function return, Vitis HLS creates an output port `ap_return` for the return value.



TIP: When the function return is specified as an AXI4-Lite interface (`s_axilite`) all the ports in the control protocol are bundled into the `s_axilite` interface. This is a common practice for software-controllable kernels or IP when an application or software driver is used to configure and control when the block starts and stops operation. This is a requirement of XRT and the Vitis kernel flow.

The `ap_ctrl_hs` block-level control protocol is the default for the Vivado IP flow. [Interfaces for Vivado IP Flow](#) shows the resulting RTL ports and behavior when Vitis HLS implements `ap_ctrl_hs` on a function.

The `ap_ctrl_chain` control protocol is the default for the Vitis kernel flow as explained in [Interfaces for Vitis Kernel Flow](#). It 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 Vitis HLS blocks together.

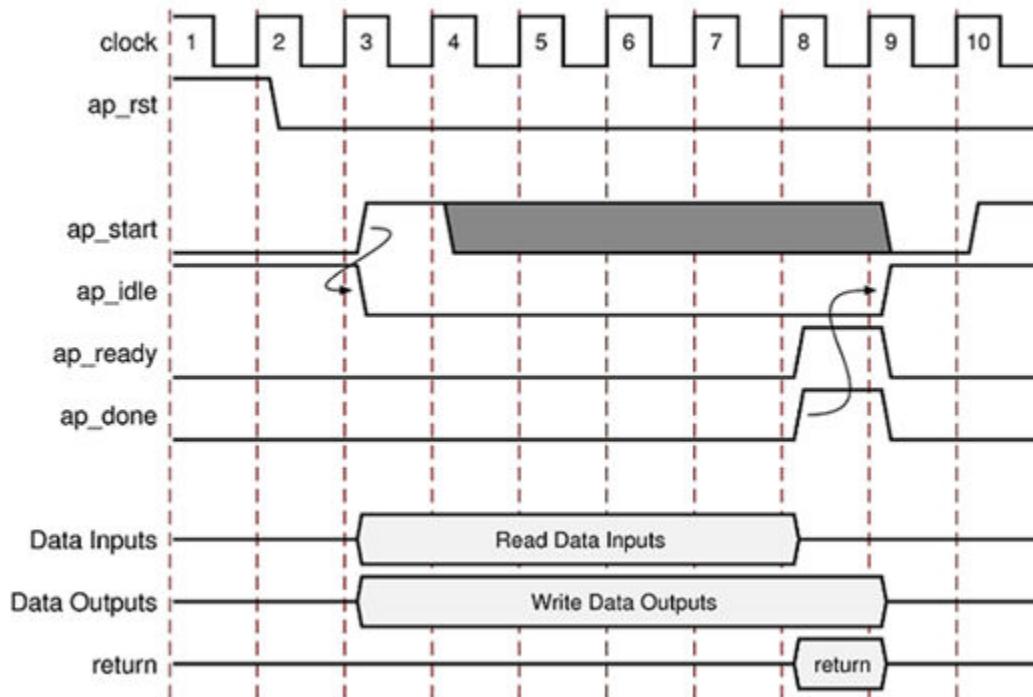


TIP: Refer to [Supported Kernel Execution Models](#) for more information on how XRT uses these control protocols.

ap_ctrl_hs

The following figure shows the behavior of the block-level handshake signals created by the ap_ctrl_hs control protocol for a non-pipelined design.

Figure 79: Behavior of ap_ctrl_hs Interface



After reset, the following occurs:

1. The block waits for ap_start to go High before it begins operation.
2. Output ap_idle goes Low immediately to indicate the design is no longer idle.
3. The ap_start signal must remain High until ap_ready goes High. Once ap_ready goes High:
 - If ap_start remains High the design will start the next transaction.
 - If ap_start is taken Low, the design will complete the current transaction and halt operation.
4. Data can be read on the input ports.
5. Data can be written to the output ports.

Note: The input and output ports can also specify a port-level I/O protocol that is independent of the control protocol. For details, see [Port-Level I/O Protocols](#).

6. Output ap_done goes High when the block completes operation.

Note: If there is an `ap_return` port, the data on this port is valid when `ap_done` is High. Therefore, the `ap_done` signal also indicates when the data on output `ap_return` is valid.

7. When the design is ready to accept new inputs, the `ap_ready` signal goes High. Following is additional information about the `ap_ready` signal:
 - The `ap_ready` signal is inactive until the design starts operation.
 - In non-pipelined designs, the `ap_ready` signal is asserted at the same time as `ap_done`.
 - In pipelined designs, the `ap_ready` signal might go High at any cycle after `ap_start` is sampled High. This depends on how the design is pipelined.
 - If the `ap_start` signal is Low when `ap_ready` is High, the design executes until `ap_done` is High and then stops operation.
 - If the `ap_start` signal is High when `ap_ready` is High, the next transaction starts immediately, and the design continues to operate.
8. The `ap_idle` signal indicates when the design is idle and not operating. Following is additional information about the `ap_idle` signal:
 - If the `ap_start` signal is Low when `ap_ready` is High, the design stops operation, and the `ap_idle` signal goes High one cycle after `ap_done`.
 - If the `ap_start` signal is High when `ap_ready` is High, the design continues to operate, and the `ap_idle` signal remains Low.

`ap_ctrl_chain`

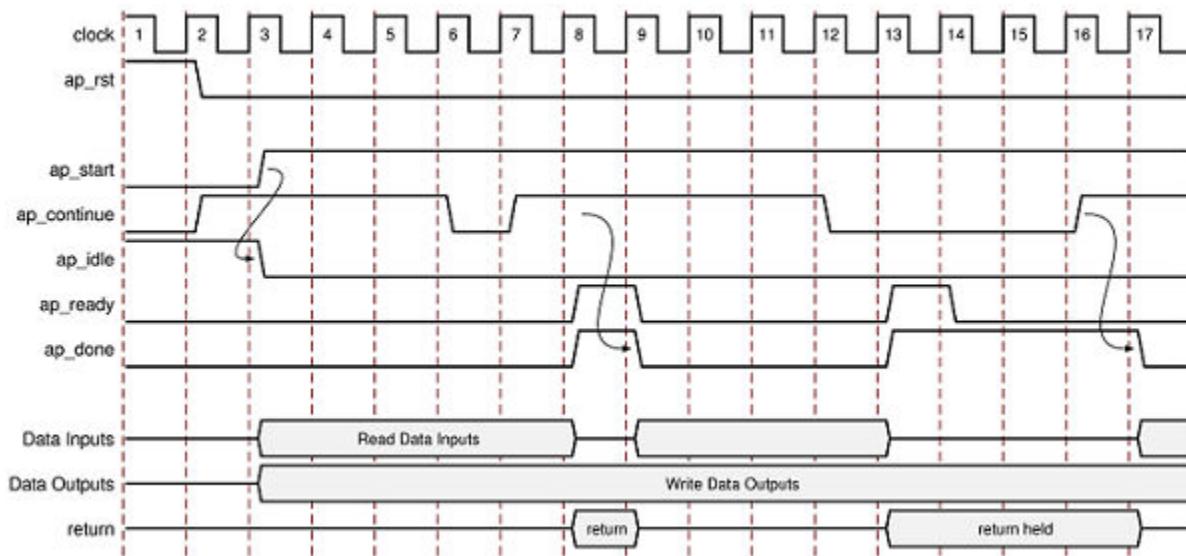
The `ap_ctrl_chain` control protocol is similar to the `ap_ctrl_hs` protocol but provides an additional input port named `ap_continue`. An active-High `ap_continue` signal indicates that the downstream block that consumes the output data is ready for new data inputs. If the downstream block is not able to consume new data inputs, the `ap_continue` signal is Low, which prevents upstream blocks from generating additional data.

The `ap_ready` port of the downstream block can directly drive the `ap_continue` port. Following is additional information about the `ap_continue` port:

- If the `ap_continue` signal is High when `ap_done` is High, the design continues operating. The behavior of the other block-level control signals is identical to those described in the `ap_ctrl_hs` block-level I/O protocol.
- If the `ap_continue` signal is Low when `ap_done` is High, the design stops operating, the `ap_done` signal remains High, and data remains valid on the `ap_return` port if the `ap_return` port is present.

In the following figure, the first transaction completes, and the second transaction starts immediately because `ap_continue` is High when `ap_done` is High. However, the design halts at the end of the second transaction until `ap_continue` is asserted High.

Figure 80: Behavior of ap_ctrl_chain Interface



ap_ctrl_none

If you specify the `ap_ctrl_none` control protocol, the handshake signal ports (`ap_start`, `ap_idle`, `ap_ready`, and `ap_done`) are not created. You can use this protocol to create a block without control signals as used in data driven kernels.



IMPORTANT! If you use the `ap_ctrl_none` control protocol in your design, you must meet at least one of the conditions for C/RTL co-simulation as described in [Interface Synthesis Requirements](#) to verify the RTL design. If at least one of these conditions is not met, C/RTL co-simulation halts with the following message:

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs:
(1)
combinational designs; (2) pipelined design with task interval of 1;
(3) designs with
array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

Managing Interfaces with SSI Technology Devices

Certain Xilinx devices use stacked silicon interconnect (SSI) technology. In these devices, the total available resources are divided over multiple super logic regions (SLRs). The connections between SLRs use super long line (SSL) routes. SSL routes incur delays costs that are typically greater than standard FPGA routing. To ensure designs operate at maximum performance, use the following guidelines:

- Register all signals that cross between SLRs at both the SLR output and SLR input.
- You do not need to register a signal if it enters or exits an SLR via an I/O buffer.
- Ensure that the logic created by Vitis HLS fits within a single SLR.

Note: When you select an SSI technology device as the target technology, the utilization report includes details on both the SLR usage and the total device usage.

If the logic is contained within a single SLR device, Vitis HLS provides a `-register_all_io` option to the `config_rtl` command. If the option is enabled, all inputs and outputs are registered. If disabled, none of the inputs or outputs are registered.

Optimization Techniques in Vitis HLS

This section outlines the various optimization techniques you can use to direct Vitis HLS to produce a micro-architecture that satisfies the desired performance and area goals. Using Vitis HLS, you can apply different optimization directives to the design, including:

- Pipelining tasks, allowing the next execution of the task to begin before the current execution is complete.
- Specifying a target latency for the completion of functions, loops, and regions.
- Specifying a limit on the number of resources used.
- Overriding the inherent or implied dependencies in the code to permit specific operations. For example, if it is acceptable to discard or ignore the initial data values, such as in a video stream, allow a memory read before write if it results in better performance.
- Specifying the I/O protocol to ensure function arguments can be connected to other hardware blocks with the same I/O protocol.

Note: Vitis HLS automatically determines the I/O protocol used by any sub-functions. You *cannot* control these ports except to specify whether the port is registered.

The optimizations techniques are presented in the context of how they are typically applied to a design:

- **Optimizing for Throughput** presents primary optimizations in the order in which they are typically used: pipeline the tasks to improve performance, improve the flow of data between tasks, and optimize structures to improve address issues which may limit performance.
- **Optimizing for Latency** uses the techniques of latency constraints and the removal of loop transitions to reduce the number of clock cycles required to complete.
- **Optimizing for Area** focuses on how operations are implemented - controlling the number of operations and how those operations are implemented in hardware - is the principal technique for improving the area.
- **Optimizing Logic** discusses optimizations affecting the implementation of the RTL.

You can add optimization directives directly into the source code as compiler pragmas using various HLS pragmas, or you can use `Tcl set_directive` commands to apply optimization directives in a `Tcl` script to be used by a solution during compilation as discussed in [Adding Pragmas and Directives](#). The following table lists the optimization directives provided by Vitis HLS as either pragma or `Tcl` directive.

Table 19: Vitis HLS Optimization Directives

Directive	Description
AGGREGATE	The AGGREGATE pragma is used for grouping all the elements of a struct into a single wide vector to allow all members of the struct to be read and written to simultaneously.
ALIAS	The ALIAS pragma enables data dependence analysis in Vitis HLS by defining the distance between pointers in the buffer.
LOCATION	Specify a limit for the number of operations, implementations, or functions used. This can force the sharing or hardware resources and may increase latency.
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
ARRAY_RESHAPE	Reshape an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM.
BIND_OP	Define a specific implementation for an operation in the RTL.
BIND_STORAGE	Define a specific implementation for a storage element, or memory, in the RTL.
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to optimize throughput and/or latency.
DEPENDENCE	Used to provide additional information that can overcome loop-carried dependencies and allow loops to be pipelined (or pipelined with lower intervals).
DISAGGREGATE	Break a struct down into its individual elements.
EXPRESSION_BALANCE	Allows automatic expression balancing to be turned off.
INLINE	Inlines a function, removing function hierarchy at this level. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
INTERFACE	Specifies how RTL ports are created from the function description.
LATENCY	Allows a minimum and maximum latency constraint to be specified.
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop with improved latency.
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.
LOOP_TRIPCOUNT	Used for loops which have variables bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.
OCCURRENCE	Used when pipelining functions or loops, to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop.
PIPELINE	Reduces the initiation interval by allowing the overlapped execution of operations within a loop or function.
PROTOCOL	This command specifies a region of code, a protocol region, in which no clock operations will be inserted by Vitis HLS unless explicitly specified in the code.
RESET	This directive is used to add or remove reset on a specific state variable (global or static).
STABLE	Indicates that a variable input or output of a dataflow region can be ignored when generating the synchronizations at entry and exit of the dataflow region.
STREAM	Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization. When using <code>hls::stream</code> , the STREAM optimization directive is used to override the configuration of the <code>hls::stream</code> .

Table 19: Vitis HLS Optimization Directives (cont'd)

Directive	Description
TOP	The top-level function for synthesis is specified in the project settings. This directive may be used to specify any function as the top-level for synthesis. This then allows different solutions within the same project to be specified as the top-level function for synthesis without needing to create a new project.
UNROLL	Unroll for-loops to create multiple instances of the loop body and its instructions that can then be scheduled independently.

In addition to the optimization directives, Vitis HLS provides a number of configuration commands that can influence the performance of synthesis results. Details on using configurations commands can be found in [Setting Configuration Options](#). The following table reflects some of these commands.

Table 20: Vitis HLS Configurations

GUI Directive	Description
Config Array Partition	Determines how arrays are partitioned, including global arrays and if the partitioning impacts array ports.
Config Compile	Controls synthesis specific optimizations such as the automatic loop pipelining and floating point math optimizations.
Config Dataflow	Specifies the default memory channel and FIFO depth in dataflow optimization.
Config Interface	Controls I/O ports not associated with the top-level function arguments and allows unused ports to be eliminated from the final RTL.
Config Op	Configures the default latency and implementation of specified operations.
Config RTL	Provides control over the output RTL including file and module naming, reset style and FSM encoding.
Config Schedule	Determines the effort level to use during the synthesis scheduling phase and the verbosity of the output messages
Config Storage	Configures the default latency and implementation of specified storage types.
Config Unroll	Configures the default tripcount threshold for unrolling loops.

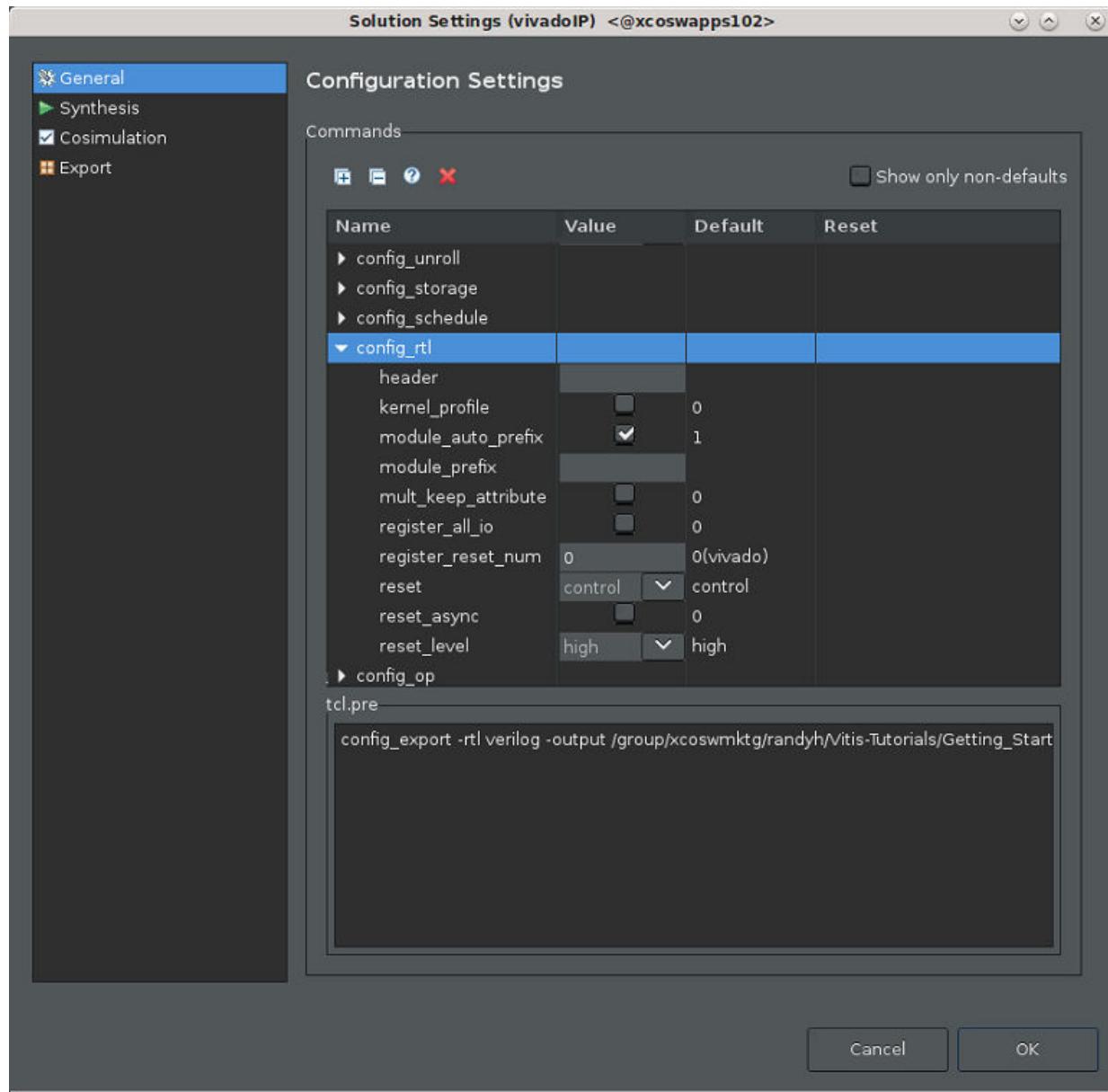
Controlling the Reset Behavior

The reset port is used in an FPGA to return the registers and block RAM connected to the reset port to an initial value any time the reset signal is applied. Typically the most important aspect of RTL configuration is selecting the reset behavior.

Note: When discussing reset behavior it is important to understand the difference between initialization and reset. Refer to [Initialization Behavior](#) for more information.

The presence and behavior of the RTL reset port is controlled using the `config_rtl` command, as shown in the following figure. You can access this command by selecting the **Solution → Solution Settings** menu command.

Figure 81: RTL Configurations



The reset settings include the ability to set the polarity of the reset and whether the reset is synchronous or asynchronous but more importantly it controls, through the **reset** option, which registers are reset when the reset signal is applied.



IMPORTANT! When AXI4 interfaces are used on a design the reset polarity is automatically changed to active-Low irrespective of the setting in the `config_rtl` configuration. This is required by the AXI4 standard.

The **reset** option has four settings:

- **none:** No reset is added to the design.

- **control:** This is the default and ensures all control registers are reset. Control registers are those used in state machines and to generate I/O protocol signals. This setting ensures the design can immediately start its operation state.
- **state:** This option adds a reset to control registers (as in the **control** setting) plus any registers or memories derived from static and global variables in the C/C++ code. This setting ensures static and global variable initialized in the C/C++ code are reset to their initialized value after the reset is applied.
- **all:** This adds a reset to all registers and memories in the design.

Finer grain control over reset is provided through the RESET pragma or directive. Static and global variables can have a reset added through the RESET directive. Variables can also be removed from those being reset by using the RESET directive's `off` option.



IMPORTANT! It is important when using the `reset state` or `all` options to consider the effect on resetting arrays as discussed in [Initializing and Resetting Arrays](#).

Initialization Behavior

In C/C++, variables defined with the `static` qualifier and those defined in the global scope are initialized to zero, by default. These variables may optionally be assigned a specific initial value. For these initialized variables, the value in the C/C++ code is assigned at compile time (at time zero) and never again. In both cases, the initial value is implemented in the RTL.

- During RTL simulation the variables are initialized with the same values as the C/C++ code.
- The variables are also initialized in the bitstream used to program the FPGA. When the device powers up, the variables will start in their initialized state.

In the RTL, although the variables start with the same initial value as the C/C++ code, there is no way to force the variable to return to this initial state. To restore the initial state, variables must be implemented with a reset signal.



IMPORTANT! Top-level function arguments can be implemented in an AXI4-Lite interface. Because there is no way to provide an initial value in C/C++ for function arguments, these variables cannot be initialized in the RTL as doing so would create an RTL design with different functional behavior from the C/C++ code which would fail to verify during C/RTL co-simulation.

Initializing and Resetting Arrays

Arrays are often defined as static variables, which implies all elements are initialized to zero; and arrays are typically implemented as block RAM. When reset options `state` or `all` are used, it forces all arrays implemented as block RAM to be returned to their initialized state after reset. This may result in two very undesirable conditions in the RTL design:

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

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

Alternatively, you can use the `state` reset mode, and use the `RESET` directive `off` option to identify individual static or global variables to remove the reset from.

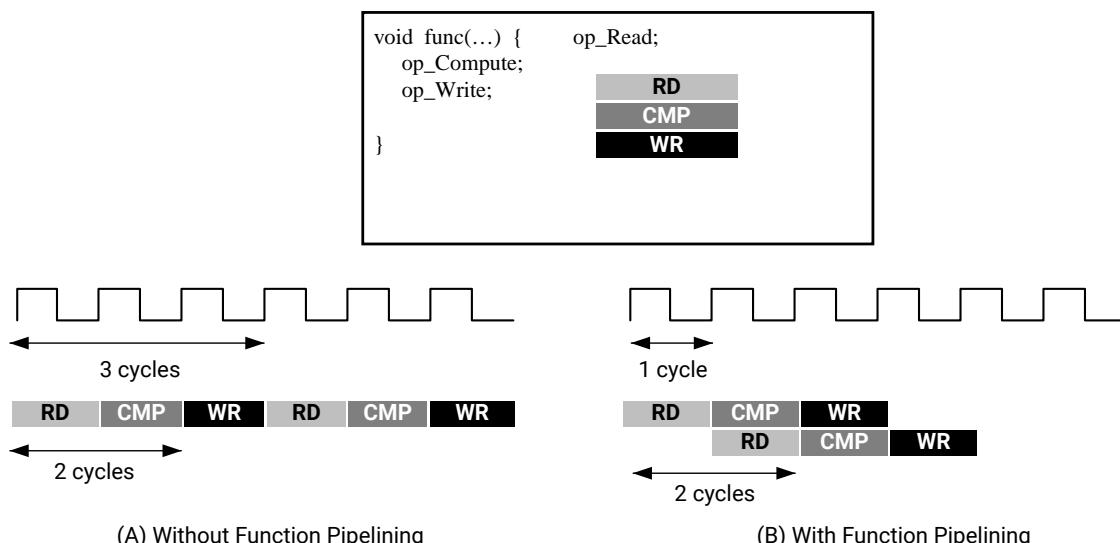
Optimizing for Throughput

Use the following optimizations to improve throughput or reduce the initiation interval.

Function and Loop Pipelining

Pipelining allows operations to happen concurrently: each execution step does not have to complete all operations before it begins the next operation. Pipelining is applied to functions and loops. The throughput improvements in function pipelining are shown in the following figure.

Figure 82: Function Pipelining Behavior



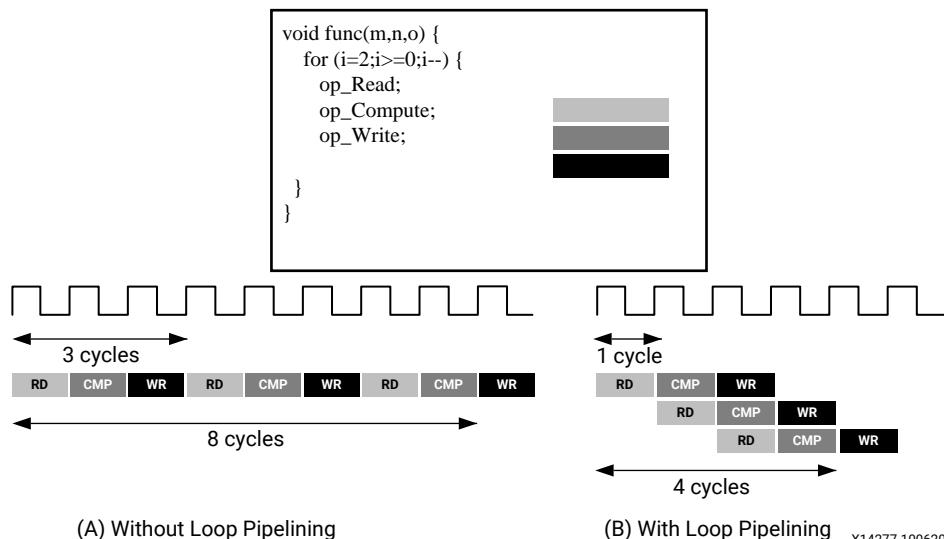
X14269-100620

Without pipelining, the function in the above example reads an input every 3 clock cycles and outputs a value after 2 clock cycles. The function has an initiation interval (II) of 3 and a latency of 3. With pipelining, for this example, a new input is read every cycle (II=1) with no change to the output latency.

Loop pipelining allows the operations in a loop to be implemented in an overlapping manner. In the following figure, (A) shows the default sequential operation where there are 3 clock cycles between each input read (II=3), and it requires 8 clock cycles before the last output write is performed.

In the pipelined version of the loop shown in (B), a new input sample is read every cycle (II=1) and the final output is written after only 4 clock cycles: substantially improving both the II and latency while using the same hardware resources.

Figure 83: Loop Pipelining



Functions or loops are pipelined using the PIPELINE directive. The directive is specified in the region that constitutes the function or loop body. The initiation interval defaults to 1 if not specified but may be explicitly specified.

Pipelining is applied only to the specified region and not to the hierarchy below. However, all loops in the hierarchy below are automatically unrolled. Any sub-functions in the hierarchy below the specified function must be pipelined individually. If the sub-functions are pipelined, the pipelined functions above it can take advantage of the pipeline performance. Conversely, any sub-function below the pipelined top-level function that is not pipelined might be the limiting factor in the performance of the pipeline.

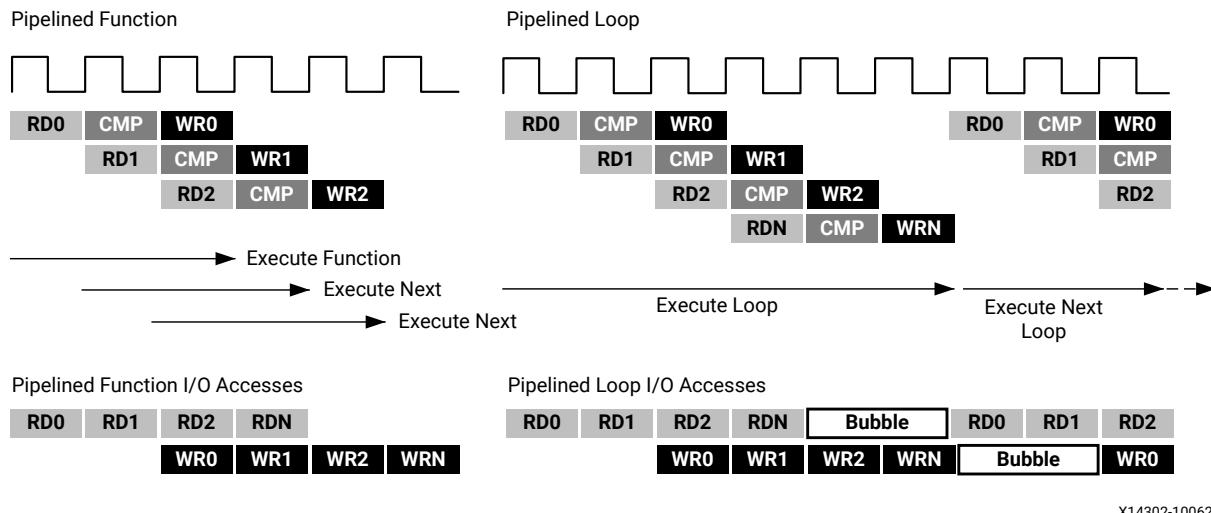
There is a difference in how pipelined functions and loops behave.

- In the case of functions, the pipeline runs forever and never ends.

- In the case of loops, the pipeline executes until all iterations of the loop are completed.

This difference in behavior is summarized in the following figure.

Figure 84: Function and Loop Pipelining Behavior



X14302-100620

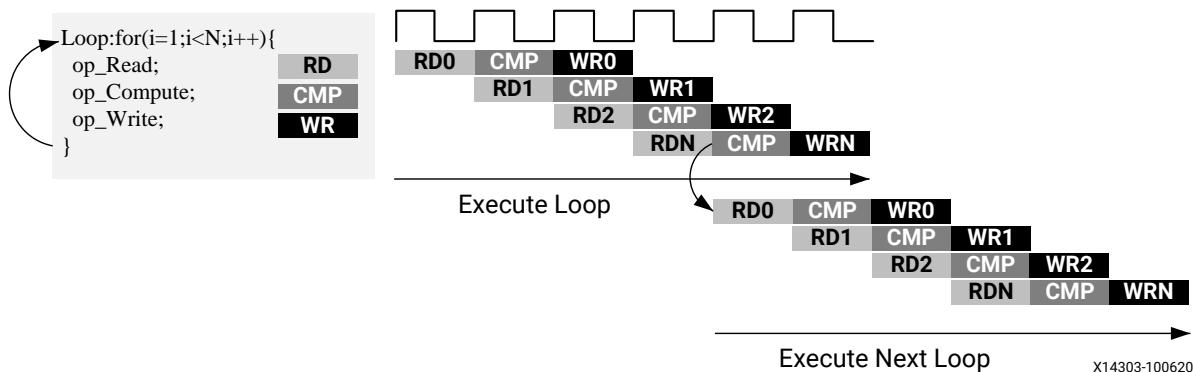
The difference in behavior impacts how inputs and outputs to the pipeline are processed. As seen in the figure above, a pipelined function will continuously read new inputs and write new outputs. By contrast, because a loop must first finish all operations in the loop before starting the next loop, a pipelined loop causes a “bubble” in the data stream; that is, a point when no new inputs are read as the loop completes the execution of the final iterations, and a point when no new outputs are written as the loop starts new loop iterations.

Rewinding Pipelined Loops for Performance

To avoid issues shown in the previous figure ([Function and Loop Pipelining](#)), the PIPELINE pragma has an optional command `rewind`. This command enables the overlap of the execution of successive calls to the loop, when this loop is the outermost construct of the top function or of a dataflow process (and the dataflow region is executed multiple times).

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

Figure 85: Loop Pipelining with Rewind Option

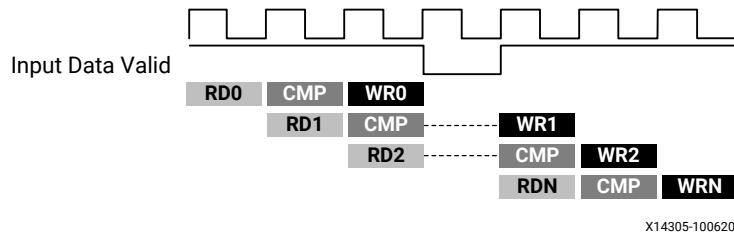


Note: If a loop is used around a DATAFLOW region, Vitis HLS automatically implements it to allow successive executions to overlap. See [Exploiting Task Level Parallelism: Dataflow Optimization](#) for more information.

Flushing Pipelines

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

Figure 86: Loop Pipelining with Stall



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

The default style of pipelining implemented by Vitis HLS is defined by the `config_compile - pipeline_style` command. You can specify stalling pipelines (stp), or free-running flushing pipelines (frp) to be used throughout the design. You can also define a third type of flushable pipeline (flp) with the `PIPELINE` pragma or directive, using the `enable_flush` option. This option applies to the specific scope of the pragma or directive only, and does not change the global default assigned by `config_compile`.

The three types of pipelines available in the tool are summarized in the following table:

Table 21: Pipeline Types

Name	Stalled Pipeline (default)	Free-Running/ Flushable Pipeline	Flushable Pipeline
Global Setting	config_compile - pipeline_style stp (default)	config_compile - pipeline_style frp	N/A
Pragma/Directive	#HLS pragma pipeline	N/A	#HLS pragma pipeline enable_flush
Advantages	<ul style="list-style-type: none"> Default pipeline. No usage constraints. Typically the lowest overall resource usage. 	<ul style="list-style-type: none"> Better timing due to <ul style="list-style-type: none"> Less fanout Simpler pipeline control logic Flushable 	<ul style="list-style-type: none"> Flushable
Disadvantages	<ul style="list-style-type: none"> Not flushable, hence it can: <ul style="list-style-type: none"> Cause more deadlocks in dataflow Prevent already computed outputs from being delivered, if the inputs to the next iterations are missing Timing issues due to high fanout on pipeline controls 	<ul style="list-style-type: none"> Moderate resource increase due to FIFOs added on outputs Usage constraints: <ul style="list-style-type: none"> Mainly used for dataflow internal processes M-AXI not supported 	<ul style="list-style-type: none"> Can have larger II Greater resource usage due to less sharing($II > 1$) Usage constraints: <ul style="list-style-type: none"> Function pipeline only
Use cases	<ul style="list-style-type: none"> When there is no timing issue due to high fanout on pipeline control When flushable is not required (such as no performance or deadlock issue due to stall) 	<ul style="list-style-type: none"> When you need better timing due to fanout to register enables from pipeline control When flushable is required for better performance or avoiding deadlock 	<ul style="list-style-type: none"> When flushable is required for better performance or avoiding deadlock

Automatic Loop Pipelining

The `config_compile` configuration enables loops to be pipelined automatically based on the iteration count. This configuration is accessed through the menu **Solution**→**Solution Setting**→**General**→**Add**→`config_compile`.

The `pipeline_loops` option sets the iteration limit. All loops with an iteration count below this limit are automatically pipelined. The default is 64.

Given the following example code:

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

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

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

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



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

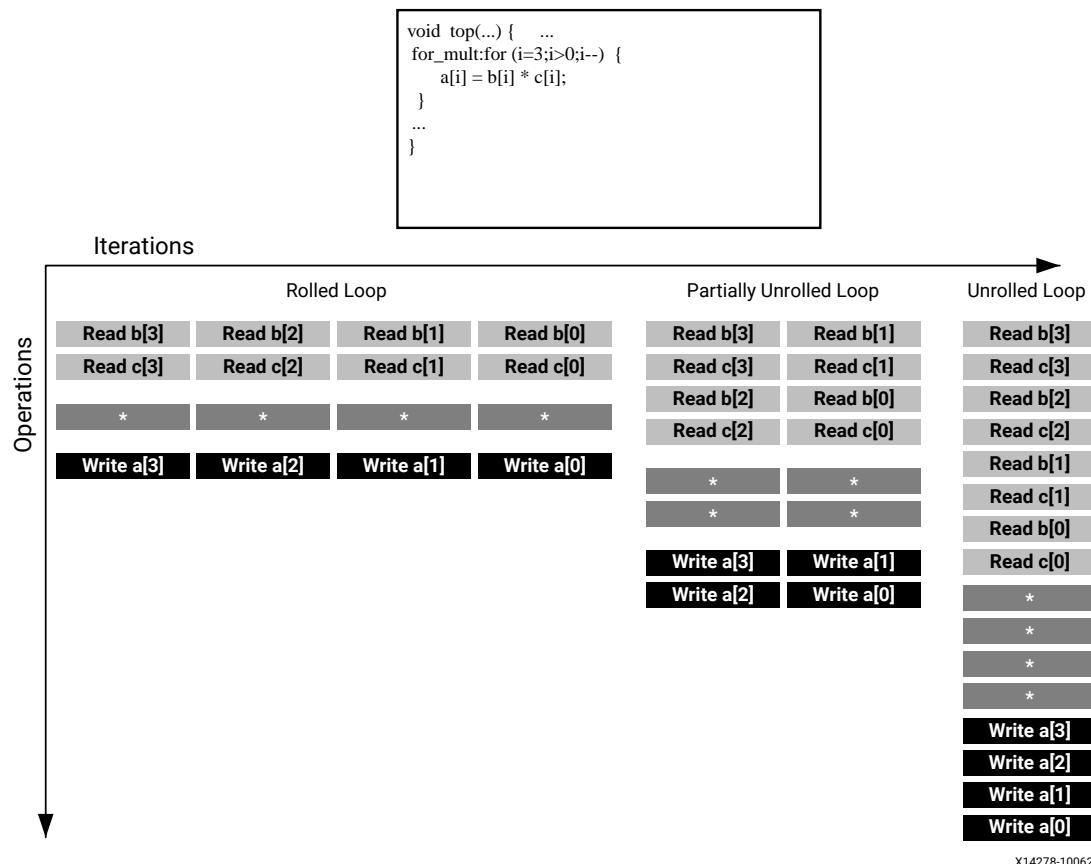
Unrolling Loops to Improve Pipelining

By default, loops are kept rolled in Vitis HLS. These rolled loops generate a hardware resource which is used by each iteration of the loop. While this creates a resource efficient block, it can sometimes be a performance bottleneck.

Vitis HLS provides the ability to unroll or partially unroll FOR loops using the `UNROLL` pragma or directive.

The following figure shows both the advantages of loop unrolling and the implications that must be considered when unrolling loops. This example assumes the arrays `a[i]`, `b[i]`, and `c[i]` are mapped to block RAMs. This example shows how easy it is to create many different implementations by the simple application of loop unrolling.

Figure 87: Loop Unrolling Details



X14278-100620

- Rolled Loop:** When the loop is rolled, each iteration is performed in separate clock cycles. This implementation takes four clock cycles, only requires one multiplier and each block RAM can be a single-port block RAM.
- Partially Unrolled Loop:** In this example, the loop is partially unrolled by a factor of 2. This implementation required two multipliers and dual-port RAMs to support two reads or writes to each RAM in the same clock cycle. This implementation does however only take 2 clock cycles to complete: half the initiation interval and half the latency of the rolled loop version.
- Unrolled loop:** In the fully unrolled version all loop operation can be performed in a single clock cycle. This implementation however requires four multipliers. More importantly, this implementation requires the ability to perform 4 reads and 4 write operations in the same clock cycle. Because a block RAM only has a maximum of two ports, this implementation requires the arrays be partitioned.

To perform loop unrolling, you can apply the UNROLL directives to individual loops in the design. Alternatively, you can apply the UNROLL directive to a function, which unrolls all loops within the scope of the function.

If a loop is completely unrolled, all operations will be performed in parallel if data dependencies and resources allow. If operations in one iteration of the loop require the result from a previous iteration, they cannot execute in parallel but will execute as soon as the data is available. A completely unrolled and fully optimized loop will generally involve multiple copies of the logic in the loop body.

The following example code demonstrates how loop unrolling can be used to create an optimized design. In this example, the data is stored in the arrays as interleaved channels. If the loop is pipelined with $\text{II}=1$, each channel is only read and written every eighth block cycle.

```
// Array Order : 0 1 2 3 4 5 6 7 8 9 10 etc. 16
etc...
// Sample Order: A0 B0 C0 D0 E0 F0 G0 H0 A1 B1 C2 etc. A2
etc...
// Output Order: A0 B0 C0 D0 E0 F0 G0 H0 A0+A1 B0+B1 C0+C2 etc. A0+A1+A2
etc...

#define CHANNELS 8
#define SAMPLES 400
#define N CHANNELS * SAMPLES

void foo (dout_t d_out[N], din_t d_in[N]) {
    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];

    // Accumulate each channel
    For_Loop: for (i=0;i<N;i++) {
        rem=i%CHANNELS;
        acc[rem] = acc[rem] + d_in[i];
        d_out[i] = acc[rem];
    }
}
```

Partially unrolling the loop by a factor of 8 will allow each of the channels (every eighth sample) to be processed in parallel (if the input and output arrays are also partitioned in a cyclic manner to allow multiple accesses per clock cycle). If the loop is also pipelined with the rewind option, this design will continuously process all 8 channels in parallel if called in a pipelined fashion (that is, either at the top, or within a dataflow region).

```
void foo (dout_t d_out[N], din_t d_in[N]) {
#pragma HLS ARRAY_PARTITION variable=d_i type=cyclic factor=8 dim=1
#pragma HLS ARRAY_PARTITION variable=d_o type=cyclic factor=8 dim=1

    int i, rem;

    // Store accumulated data
    static dacc_t acc[CHANNELS];

    // Accumulate each channel
    For_Loop: for (i=0;i<N;i++) {
#pragma HLS PIPELINE rewind
#pragma HLS UNROLL factor=8
```

```

rem=i%CHANNELS;
acc[rem] = acc[rem] + d_in[i];
d_out[i] = acc[rem];
}
}

```

Partial loop unrolling does not require the unroll factor to be an integer multiple of the maximum iteration count. Vitis HLS adds an exit check to ensure partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```

for(int i = 0; i < N; i++) {
    a[i] = b[i] + c[i];
}

```

Loop unrolling by a factor of 2 effectively transforms the code to look like the following example where the `break` construct is used to ensure the functionality remains the same:

```

for(int i = 0; i < N; i += 2) {
    a[i] = b[i] + c[i];
    if (i+1 >= N) break;
    a[i+1] = b[i+1] + c[i+1];
}

```

Because `N` is a variable, Vitis HLS might not be able to determine its maximum value (it could be driven from an input port). If the unrolling factor, which is 2 in this case, is an integer factor of the maximum iteration count `N`, the `skip_exit_check` option removes the exit check and associated logic. The effect of unrolling can now be represented as:

```

for(int i = 0; i < N; i += 2) {
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
}

```

This helps minimize the area and simplify the control logic.

Addressing Failure to Pipeline

When a function is pipelined, all loops in the hierarchy below are automatically unrolled. This is a requirement for pipelining to proceed. If a loop has variable bounds it cannot be unrolled. This will prevent the function from being pipelined.

Static Variables

Static variables are used to keep data between loop iterations, often resulting in registers in the final implementation. If this is encountered in pipelined functions, Vitis HLS might not be able to optimize the design sufficiently, which would result in initiation intervals longer than required.

The following is a typical example of this situation:

```
function_foo()
{
    static bool change = 0
    if (condition_xyz){
        change = x; // store
    }
    y = change; // load
}
```

If Vitis HLS cannot optimize this code, the stored operation requires a cycle and the load operation requires an additional cycle. If this function is part of a pipeline, the pipeline has to be implemented with a minimum initiation interval of 2 as the static change variable creates a loop-carried dependency.

One way the user can avoid this is to rewrite the code, as shown in the following example. It ensures that only a read or a write operation is present in each iteration of the loop, which enables the design to be scheduled with II=1.

```
function_readstream()
{
    static bool change = 0
    bool change_temp = 0;
    if (condition_xyz)
    {
        change = x; // store
        change_temp = x;
    }
    else
    {
        change_temp = change; // load
    }
    y = change_temp;
}
```

Partitioning Arrays to Improve Pipelining

A common issue when pipelining functions is the following message:

```
INFO: [SCHED 204-61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
WARNING: [SCHED 204-69] The resource limit of core:RAM:mem:p0 is 1, current
assignments:
WARNING: [SCHED 204-69]      'load' operation ('mem_load', bottleneck.c:62)
on array
'mem',
WARNING: [SCHED 204-69] The resource limit of core:RAM:mem:p1 is 1, current
assignments:
WARNING: [SCHED 204-69]      'load' operation ('mem_load_1',
bottleneck.c:62) on array
'mem',
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

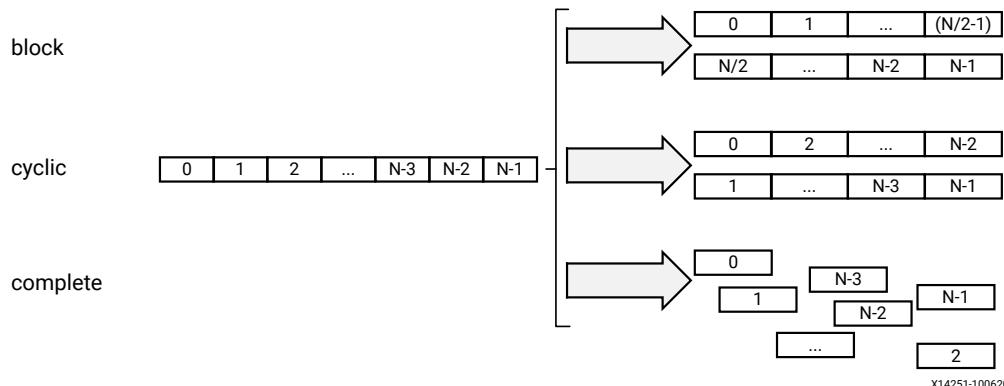
In this example, Vitis HLS states it cannot reach the specified initiation interval (II) of 1 because it cannot schedule a `load` (read) operation (`mem_load_2`) onto the memory because of limited memory ports. The above message notes that the resource limit for "core:RAM:mem:p0 is 1" which is used by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

This issue is typically caused by arrays. Arrays are implemented as block RAM which only has a maximum of two data ports. This can limit the throughput of a read/write (or load/store) intensive algorithm. The bandwidth can be improved by splitting the array (a single block RAM resource) into multiple smaller arrays (multiple block RAMs), effectively increasing the number of ports.

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

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

Figure 88: Array Partitioning



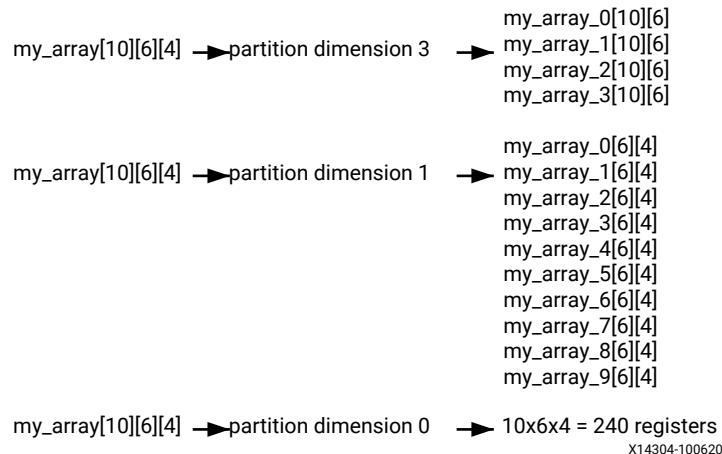
For `block` and `cyclic` partitioning the `factor` option specifies the number of arrays that are created. In the preceding figure, a factor of 2 is used, that is, the array is divided into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the final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

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

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

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

Figure 89: Partitioning Array Dimensions



Automatic Array Partitioning

The `config_array_partition` configuration determines how arrays are automatically partitioned based on the number of elements. This configuration is accessed through the menu **Solution**→**Solution Settings**→**General**→**Add**→**config_array_partition**.

Managing Pipeline Dependencies

Vitis HLS constructs a hardware datapath that corresponds to the C/C++ source code.

When there is no pipeline directive, the execution is sequential so there are no dependencies to take into account. But when the design has been pipelined, the tool needs to deal with the same dependencies as found in processor architectures for the hardware that Vitis HLS generates.

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

- A read-after-write (RAW), also called a true dependency, is when an instruction (and data it reads/uses) depends on the result of a previous operation.

- I1: $t = a * b;$
- I2: $c = t + 1;$

The read in statement I2 depends on the write of t in statement I1. If the instructions are reordered, it uses the previous value of t .

- A write-after-read (WAR), also called an anti-dependence, is when an instruction cannot update a register or memory (by a write) before a previous instruction has read the data.

- I1: $b = t + a;$
- I2: $t = 3;$

The write in statement I2 cannot execute before statement I1, otherwise the result of b is invalid.

- A write-after-write (WAW) is a dependence when a register or memory must be written in specific order otherwise other instructions might be corrupted.

- I1: $t = a * b;$
- I2: $c = t + 1;$
- I3: $t = 1;$

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

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

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

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

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

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

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

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

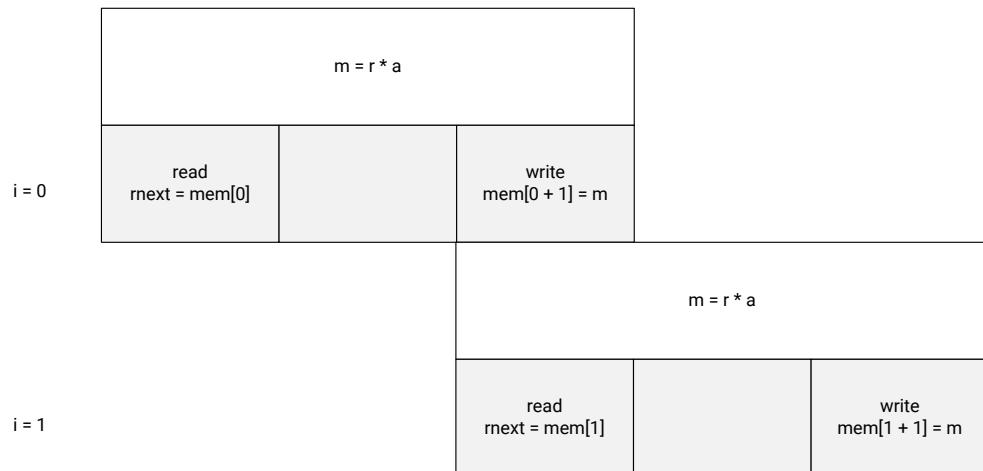
```
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint
(II = 1,
distance = 1)
between 'store' operation (top.cpp:7) of variable 'm', top.cpp:7 on array
'mem' and
'load' operation ('rnext', top.cpp:8) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: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

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

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

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

Figure 90: Dependency Example



X24685-100620

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

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

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

The scheduling warning is:

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

```

distance = 1)
between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array
'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 4, Depth: 4.

```

Observe the continued read and writes over a few iterations:

```

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

```

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

Removing False Dependencies to Improve Loop Pipelining

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

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

```

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

```

```

        old = val;
    }

    hist[old] = acc;
}

```

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

```

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

    hist[old] = acc;
}

```

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

When specifying dependencies there are two main types:

- **Inter:** Specifies the dependency is between different iterations of the same loop.
If this is specified as FALSE it allows Vitis HLS to perform operations in parallel if the pipelined or loop is unrolled or partially unrolled and prevents such concurrent operation when specified as TRUE.
- **Intra:** Specifies dependence within the same iteration of a loop, for example an array being accessed at the start and end of the same iteration.

When intra dependencies are specified as FALSE, Vitis HLS may move operations freely within the loop, increasing their mobility and potentially improving performance or area. When the dependency is specified as TRUE, the operations must be performed in the order specified.

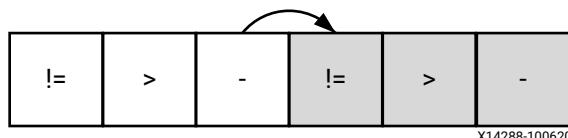
Scalar Dependencies

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

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

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

Figure 91: Scalar Dependency



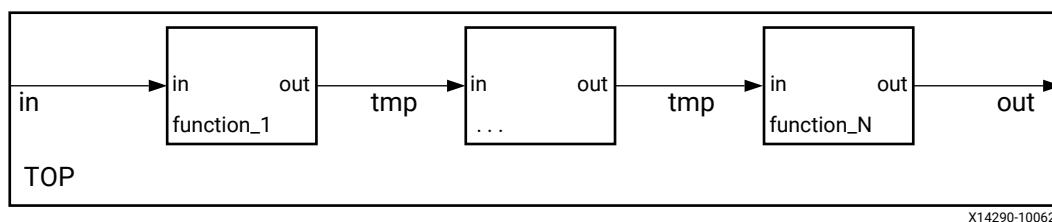
X14288-100620

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

Exploiting Task Level Parallelism: Dataflow Optimization

The dataflow optimization is useful on a set of sequential tasks (for example, functions and/or loops), as shown in the following figure.

Figure 92: Sequential Functional Description

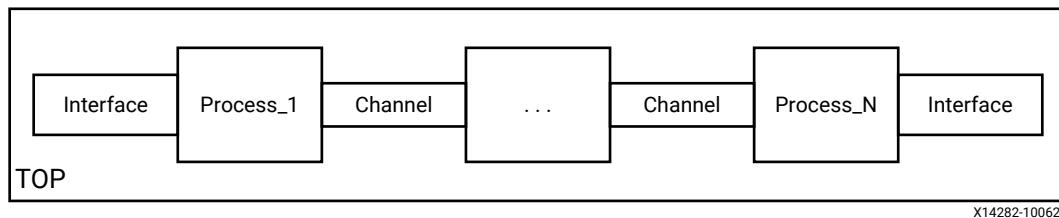


X14290-100620

The above figure shows a specific case of a chain of three tasks, but the communication structure can be more complex than shown.

Using this series of sequential tasks, dataflow optimization creates an architecture of concurrent processes, as shown below. Dataflow optimization is a powerful method for improving design throughput and latency.

Figure 93: Parallel Process Architecture

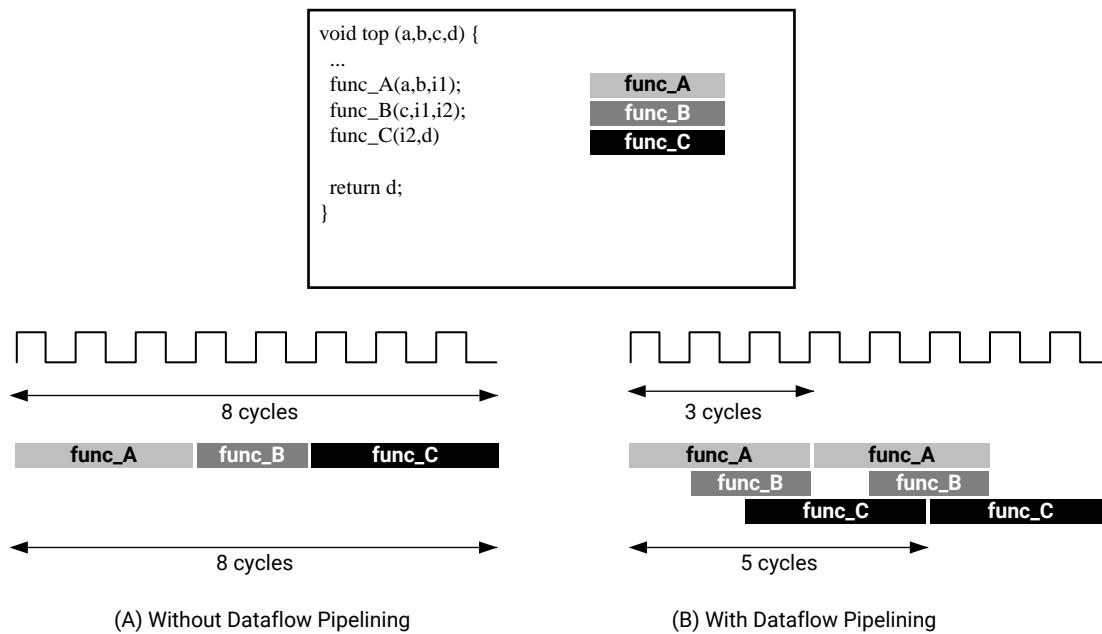


The following figure shows how dataflow optimization allows the execution of tasks to overlap, increasing the overall throughput of the design and reducing latency.

In the following figure and example, (A) represents the case without the dataflow optimization. The implementation requires 8 cycles before a new input can be processed by `func_A` and 8 cycles before an output is written by `func_C`.

For the same example, (B) represents the case when the dataflow optimization is applied. `func_A` can begin processing a new input every 3 clock cycles (lower initiation interval) and it now only requires 5 clocks to output a final value (shorter latency).

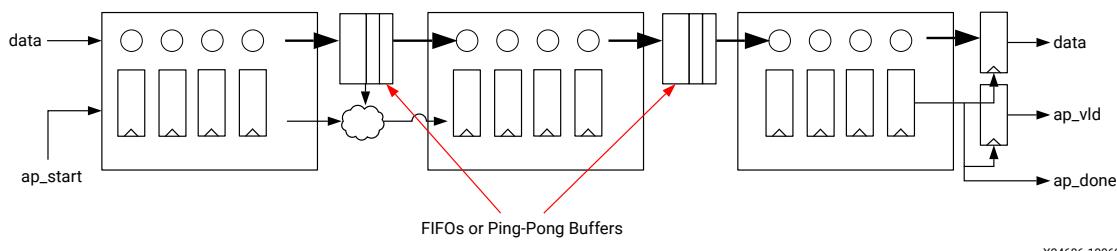
Figure 94: Dataflow Optimization



X14266-100620

This type of parallelism cannot be achieved without incurring some overhead in hardware. When a particular region, such as a function body or a loop body, is identified as a region to apply the dataflow optimization, Vitis HLS analyzes the function or loop body and creates individual channels that model the dataflow to store the results of each task in the dataflow region. These channels can be simple FIFOs for scalar variables, or ping-pong (PIPO) buffers for non-scalar variables like arrays. Each of these channels also contain signals to indicate when the FIFO or the ping-pong buffer is full or empty. These signals represent a handshaking interface that is completely data driven. By having individual FIFOs and/or ping-pong buffers, Vitis HLS frees each task to execute at its own pace and the throughput is only limited by availability of the input and output buffers. This allows for better interleaving of task execution than a normal pipelined implementation but does so at the cost of additional FIFO or block RAM registers for the ping-pong buffer, as shown in the following figure.

Figure 95: Structure Created During Dataflow Optimization



X24686-100620

Dataflow optimization potentially improves performance over a statically pipelined solution. It replaces the strict, centrally-controlled pipeline stall philosophy with more flexible and distributed handshaking architecture using FIFOs and/or ping-pong buffers (PIPOs). The replacement of the centralized control structure with a distributed one also benefits the fanout of control signals, for example register enables, which is distributed among the control structures of individual processes.

Dataflow optimization is not limited to a chain of processes, but can be used on any directed acyclic graph (DAG) structure. It can produce two different forms of overlapping: within an iteration if processes are connected with FIFOs, and across different iterations through PIPOs and FIFOs.

Canonical Forms

Vitis HLS transforms the region to apply the DATAFLOW optimization. Xilinx recommends writing the code inside this region (referred to as the *canonical region*) using canonical forms. There are two main canonical forms for the dataflow optimization:

1. The canonical form for a function where sub-functions are not inlined.

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

2. Dataflow inside a loop body.

For the for loop (where no function inside is inlined), the integral loop variable should have:

- a. Initial value declared in the loop header and set to 0.
- b. The loop condition is a positive numerical constant or constant function argument.
- c. Increment by 1.
- d. Dataflow pragma needs to be inside the loop.

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

Canonical Body

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

1. Use a local, non-static scalar or array/pointer variable, or local static stream variable. A local variable is declared inside the function body (for dataflow in a function) or loop body (for dataflow inside a loop).
2. A sequence of function calls that pass data forward (with no feedback), from a function to one that is lexically later, under the following conditions:
 - a. Variables (except scalar) can have only one reading process and one writing process.
 - b. Use write before read (producer before consumer) if you are using local variables, which then become channels.
 - c. Use read before write (consumer before producer) if you are using function arguments. Any intra-body anti-dependencies must be preserved by the design.
 - d. Function return type must be void.

- e. No loop-carried dependencies among different processes via variables.
 - Inside the canonical loop (i.e., values written by one iteration and read by a following one).
 - Among successive calls to the top function (i.e., inout argument written by one iteration and read by the following iteration).
- f. No control whatsoever is supported inside a dataflow region, except for function calls (that define processes).
 - No conditional, no loop, no return, no goto, no throw.
 - The only control supported around dataflow is:
 - Simple for loop, with unsigned integer induction variable initialized to 0, incremented by 1, and compared either with a non-negative constant or with an unsigned input of the function containing the dataflow-in-loop without any other statement in the function containing dataflow in loop, except for variable declarations. Typically only streams used in the loop body can be declared at that level.

Dataflow Checking

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

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

Dataflow Optimization Limitations

The DATAFLOW optimization optimizes the flow of data between tasks (functions and loops), and ideally pipelined functions and loops for maximum performance. It does not require these tasks to be chained, one after the other, however there are some limitations in how the data is transferred.

The following behaviors can prevent or limit the overlapping that Vitis HLS can perform with DATAFLOW optimization:

- Reading from function inputs or writing to function outputs in the middle of the dataflow region
- Single-producer-consumer violations
- Bypassing tasks and channel sizing
- Feedback between tasks
- Conditional execution of tasks

- Loops with multiple exit conditions



IMPORTANT! If any of these coding styles are present, Vitis HLS issues a message describing the situation.

Note: You can use the Dataflow viewer in the Analysis perspective to view the structure when the DATAFLOW directive is applied.

Reading from Inputs/Writing to Outputs

Reading of inputs of the function should be done at the start of the dataflow region, and writing to outputs should be done at the end of the dataflow region. Reading/writing to the ports of the function can cause the processes to be executed in sequence rather than in an overlapped fashion, adversely impacting performance.

Single-producer-consumer Violations

For Vitis HLS to perform the DATAFLOW optimization, all elements passed between tasks must follow a single-producer-consumer model. Each variable must be driven from a single task and only be consumed by a single task. In the following code example, `temp1` fans out and is consumed by both `Loop2` and `Loop3`. This violates the single-producer-consumer model.

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {
    int temp1[N];

    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        data_out1[j] = temp1[j] * 123;
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out2[k] = temp1[k] * 456;
    }
}
```

A modified version of this code uses function `Split` to create a single-producer-consumer design. The following code block example shows how the data flows with the function `Split`. The data now flows between all four tasks, and Vitis HLS can perform the DATAFLOW optimization.

```
void Split (in[N], out1[N], out2[N]) {
// Duplicated data
L1:for(int i=1;i<N;i++) {
    out1[i] = in[i];
    out2[i] = in[i];
}
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

    int temp1[N], temp2[N]. temp3[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
    }
}
```

```

Split(temp1, temp2, temp3);
Loop2: for(int j = 0; j < N; j++) {
    data_out1[j] = temp2[j] * 123;
}
Loop3: for(int k = 0; k < N; k++) {
    data_out2[k] = temp3[k] * 456;
}
}

```

Bypassing Tasks and Channel Sizing

In addition, data should generally flow from one task to another. If you bypass tasks, this can reduce the performance of the DATAFLOW optimization. In the following example, Loop1 generates the values for `temp1` and `temp2`. However, the next task, Loop2, only uses the value of `temp1`. The value of `temp2` is not consumed until *after* Loop2. Therefore, `temp2` bypasses the next task in the sequence, which can limit the performance of the DATAFLOW optimization.

```

void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {
    int temp1[N], temp2[N], temp3[N];
    Loop1: for(int i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
        temp2[i] = data_in[i] >> scale;
    }
    Loop2: for(int j = 0; j < N; j++) {
        temp3[j] = temp1[j] + 123;
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out1[k] = temp2[k] + temp3[k];
    }
}

```

In this case, you should increase the depth of the PIPO buffer used to store `temp2` to be 3, instead of the default depth of 2. This lets the buffer store the value intended for Loop3, while Loop2 is being executed. Similarly, a PIPO that bypasses two processes should have a depth of 4. Set the depth of the buffer with the `STREAM` pragma or directive:

```
#pragma HLS STREAM type=piro variable=temp2 depth=3
```



IMPORTANT! Channel sizing can also similarly affect performance. Having mismatched FIFO/PIPO depths can inadvertently cause synchronization points inside the dataflow region because of back pressure from the FIFO/PIPO.

Feedback between Tasks

Feedback occurs when the output from a task is consumed by a previous task in the DATAFLOW region. Feedback between tasks is not recommended in a DATAFLOW region. When Vitis HLS detects feedback, it issues a warning, depending on the situation, and might not perform the DATAFLOW optimization.

However, DATAFLOW can support feedback when used with `hls::streams`. The following example demonstrates this exception.

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

void firstProc(hls::stream<int> &forwardOUT, hls::stream<int> &backwardIN) {
    static bool first = true;
    int fromSecond;

    //Initialize stream
    if (first)
        fromSecond = 10; // Initial stream value
    else
        //Read from stream
        fromSecond = backwardIN.read(); //Feedback value
    first = false;

    //Write to stream
    forwardOUT.write(fromSecond*2);
}

void secondProc(hls::stream<int> &forwardIN, hls::stream<int> &backwardOUT)
{
    backwardOUT.write(forwardIN.read() + 1);
}

void top(...){
#pragma HLS dataflow
    hls::stream<int> forward, backward;
    firstProc(forward, backward);
    secondProc(forward, backward);
}
```

In this simple design, when `firstProc` is executed, it uses 10 as an initial value for input. Because `hls::streams` do not support an initial value, this technique can be used to provide one without violating the single-producer-consumer rule. In subsequent iterations `firstProc` reads from the `hls::stream` through the `backwardIN` interface.

`firstProc` processes the value and sends it to `secondProc`, via a stream that goes *forward* in terms of the original C++ function execution order. `secondProc` reads the value on `forwardIN`, adds 1 to it, and sends it back to `firstProc` via the feedback stream that goes *backwards* in the execution order.

From the second execution, `firstProc` uses the value read from the stream to do its computation, and the two processes can keep going forever, with both forward and feedback communication, using an initial value for the first execution.

Conditional Execution of Tasks

The DATAFLOW optimization does not optimize tasks that are conditionally executed. The following example highlights this limitation. In this example, the conditional execution of Loop1 and Loop2 prevents Vitis HLS from optimizing the data flow between these loops, because the data does not flow from one loop into the next.

```
void foo(int data_in[N], int data_out[N], int sel) {
    int temp1[N], temp2[N];

    if (sel) {
        Loop1: for(int i = 0; i < N; i++) {
            temp1[i] = data_in[i] * 123;
            temp2[i] = data_in[i];
        }
    } else {
        Loop2: for(int j = 0; j < N; j++) {
            temp1[j] = data_in[j] * 321;
            temp2[j] = data_in[j];
        }
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[k] = temp1[k] * temp2[k];
    }
}
```

To ensure each loop is executed in all cases, you must transform the code as shown in the following example. In this example, the conditional statement is moved into the first loop. Both loops are always executed, and data always flows from one loop to the next.

```
void foo(int data_in[N], int data_out[N], int sel) {
    int temp1[N], temp2[N];

    Loop1: for(int i = 0; i < N; i++) {
        if (sel) {
            temp1[i] = data_in[i] * 123;
        } else {
            temp1[i] = data_in[i] * 321;
        }
    }
    Loop2: for(int j = 0; j < N; j++) {
        temp2[j] = data_in[j];
    }
    Loop3: for(int k = 0; k < N; k++) {
        data_out[k] = temp1[k] * temp2[k];
    }
}
```

Loops with Multiple Exit Conditions

Loops with multiple exit points cannot be used in a DATAFLOW region. In the following example, Loop2 has three exit conditions:

- An exit defined by the value of N; the loop will exit when $k \geq N$.

- An exit defined by the `break` statement.
- An exit defined by the `continue` statement.

```
#include "ap_int.h"
#define N 16

typedef ap_int<8> din_t;
typedef ap_int<15> dout_t;
typedef ap_uint<8> dsc_t;
typedef ap_uint<1> dsel_t;

void multi_exit(din_t data_in[N], dsc_t scale, dsel_t select, dout_t
data_out[N]) {
    dout_t temp1[N], temp2[N];
    int i,k;

    Loop1: for(i = 0; i < N; i++) {
        temp1[i] = data_in[i] * scale;
        temp2[i] = data_in[i] >> scale;
    }

    Loop2: for(k = 0; k < N; k++) {
        switch(select) {
            case 0: data_out[k] = temp1[k] + temp2[k];
            case 1: continue;
            default: break;
        }
    }
}
```

Because a loop's exit condition is always defined by the loop bounds, the use of `break` or `continue` statements will prohibit the loop being used in a DATAFLOW region.

Finally, the DATAFLOW 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 DATAFLOW optimization to the loop, the sub-function, or inline the sub-function.

You can also use `std::complex` inside the DATAFLOW region. However, they should be used with an `__attribute__((no_ctor))` as shown in the following example:

```
void proc_1(std::complex<float> (&buffer)[50], const std::complex<float>
*in);
void proc_2(hls::Stream<std::complex<float>> &fifo, const
std::complex<float> (&buffer)[50], std::complex<float> &acc);
void proc_3(std::complex<float> *out, hls::Stream<std::complex<float>>
&fifo, const std::complex<float> acc);

void top(std::complex<float> *out, const std::complex<float> *in) {
#pragma HLS DATAFLOW

    std::complex<float> acc __attribute__((no_ctor)); // Here
    std::complex<float> buffer[50] __attribute__((no_ctor)); // Here
    hls::Stream<std::complex<float>, 5> fifo; // Not here

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

Configuring Dataflow Memory Channels

Vitis HLS implements channels between the tasks as either ping-pong or FIFO buffers, depending on the access patterns of the producer and the consumer of the data:

- For scalar, pointer, and reference parameters, Vitis HLS implements the channel as a FIFO.
- If the parameter (producer or consumer) is an array, Vitis HLS implements the channel as a ping-pong buffer or a FIFO as follows:
 - If Vitis HLS determines the data is accessed in sequential order, Vitis HLS implements the memory channel as a FIFO channel with a depth that is estimated to optimize performance (but can require manual tuning in practice).
 - If Vitis HLS is unable to determine that the data is accessed in sequential order or determines the data is accessed in an arbitrary manner, Vitis HLS implements the memory channel as a ping-pong buffer, that is, as two block RAMs each defined by the maximum size of the consumer or producer array.

Note: A ping-pong buffer ensures that the channel always has the capacity to hold all samples without a loss. However, this might be an overly conservative approach in some cases.

To explicitly specify the default channel used between tasks, use the `config_dataflow` configuration. This configuration sets the default channel for all channels in a design. To reduce the size of the memory used in the channel and allow for overlapping within an iteration, you can use a FIFO. To explicitly set the depth (that is, number of elements) in the FIFO, use the `-fifo_depth` option.

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

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

When setting the depth of the FIFOs, Xilinx recommends initially setting the depth as the maximum number data values transferred (for example, the size of the array passed between tasks), confirming the design passes C/RTL co-simulation, and then reducing the size of the FIFOs and confirming C/RTL co-simulation still completes without issues. If RTL co-simulation fails, the size of the FIFO is likely too small to prevent stalling or a deadlock situation.

The Vitis HLS IDE can display a histogram of the occupation of each FIFO/PIPO buffer over time, after RTL co-simulation has been run. This can be useful to help determine the best depth for each buffer.

Specifying Arrays as Ping-Pong Buffers or FIFOs

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

```
void top ( ... ) {  
#pragma HLS dataflow  
    int A[1024];  
#pragma HLS stream off variable=A depth=3  
  
    producer(A, B, ...); // producer writes A and B  
    middle(B, C, ...); // middle reads B and writes C  
    consumer(A, C, ...); // consumer reads A and C
```

In the interface, arrays are automatically specified as streaming if an array on the top-level function interface is set as interface type ap_fifo, axis or ap_hs, it is automatically set as streaming.

Inside the design, all arrays must be specified as streaming using the STREAM directive if a FIFO is desired for the implementation.

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

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

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

Note: To use a FIFO implementation, the array must be accessed in a streaming manner.

- If the config_dataflow default_channel is set to FIFO or Vitis HLS has automatically determined the data in a DATAFLOW region is accessed in a streaming manner, any array can still be implemented as a ping-pong implementation by applying the STREAM directive to the array with the -off option.



IMPORTANT! To preserve the accesses, it might be necessary to prevent compiler optimizations (dead code elimination particularly) by using the volatile qualifier.

When an array in a DATAFLOW region is specified as streaming and implemented as a FIFO, the FIFO is typically not required to hold the same number of elements as the original array. The tasks in a DATAFLOW region consume each data sample as soon as it becomes available. The config_dataflow command with the -fifo_depth option or the STREAM directive with the -depth can be used to set the size of the FIFO to the minimum number of elements required to ensure flow of data never stalls. If the -off option is selected, the -depth option sets the depth (number of blocks) of the PIPO. The depth should be at least 2.

Specifying Compiler-Created FIFO Depth

Start Propagation

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

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

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

```
#pragma HLS DATAFLOW disable_start_propagation
```



TIP: This is required when using block control protocol `ap_ctrl_none`.

Scalar Propagation

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

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

Stable Arrays

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

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

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

In the previous example, without the `stable` pragma, and assuming that `A` is read by `proc2` as `proc2` is bypassing the tasks, there will be a performance loss.

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

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

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

Using ap_ctrl_none Inside the Dataflow

The `ap_ctrl_none` block-level I/O protocol avoids the rigid synchronization scheme implied by the `ap_ctrl_hs` and `ap_ctrl_chain` protocols. These protocols require that all processes in the region are executed exactly the same number of times in order to better match the C/C++ behavior.

However, there are situations where, for example, the intent is to have a faster process that executes more frequently to distribute work to several slower ones.

For any dataflow region (except "dataflow-in-loop"), it is possible to specify `#pragma HLS interface mode=ap_ctrl_none port=return` as long as all the following conditions are satisfied:

- The region and all the processes it contains communicates only via FIFOs (`hls::stream`, streamed arrays, AXIS); that is, excluding memories.
- All the parents of the region, up to the top level design, must fit the following requirements:
 - They must be dataflow regions (excluding "dataflow-in-loop").
 - They must all specify `ap_ctrl_none`.

This means that none of the parents of a dataflow region with `ap_ctrl_none` in the hierarchy can be:

- A sequential or pipelined FSM
- A dataflow region inside a for loop ("dataflow-in-loop")

The result of this pragma is that `ap_ctrl_chain` is not used to synchronize any of the processes inside that region. They are executed or stalled based on the availability of data in their input FIFOs and space in their output FIFOs. For example:

```
void region(...) {
#pragma HLS dataflow
#pragma HLS interface mode=ap_ctrl_none port=return
    hls::stream<int> outStream1, outStream2;
    demux(inStream, outStream1, outStream2);
    worker1(outStream1, ...);
    worker2(outStream2, ...);
```

In this example, `demux` can be executed twice as frequently as `worker1` and `worker2`. For example, it can have `ll=1` while `worker1` and `worker2` can have `ll=2`, and still achieving a global `ll=1` behavior.

Note:

- Non-blocking reads may need to be used very carefully inside processes that are executed less frequently to ensure that C/C++ simulation works.
- The pragma is applied to a *region*, not to the individual processes inside it.
- Deadlock detection must be disabled in co-simulation. This can be done with the `-disable_deadlock_detection` option in [cosim_design](#).

Improve Performance Using Stream-of-Blocks

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

Currently, Vitis HLS implements arrays written by a producer process and read by a consumer process in a dataflow region by mapping them to ping pong buffers (PIPOs). The buffer exchange for a PIPO buffer is driven by the `ap_done/ap_continue` handshake of the producer process, and by the `ap_start/ap_ready` handshake of the consumer process. In other words, the exchange occurs at the return of the producer function and the calling of the consumer function in C++.

While this ensures a concurrent communication semantic that is fully compliant with the sequential C++ execution semantics, it also implies that the consumer cannot start until the producer is done, as shown in the following code example.

```
void producer( int b[M][N], ... ) {
    for ( int i = 0; i < M; i++ )
        for ( int j = 0; j < N; j++ )
            b[i][f(j)] = ... ;
}

void consumer( int b[M][N], ... ) {
```

```

    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
            ... = b[i][g(j)] ...;
}

void top(...) {
#pragma HLS dataflow
    int b[M][N];
#pragma HLS stream off variable=b

    producer(b, ...);
    consumer(b, ...);
}

```

This can unnecessarily limit throughput if the producer generates data for the consumer in smaller blocks, for example by writing one row of the buffer output inside a nested loop, and the consumer uses the data in smaller blocks by reading one row of the buffer input inside a nested loop, as the example above does. In this example, due to the non-sequential buffer column access in the inner loop you cannot simply stream the array `b`. However, the row access in the outer loop is sequential thus supporting `hls::stream_of_blocks` communication where each block is a 1-dimensional array of size `N`.

The main purpose of the `hls::stream_of_blocks` feature is to provide PIPO-like functionality, but with user-managed explicit synchronization, accesses, and better coding style. Stream-of-blocks lets you avoid the use of dataflow in a loop containing the producer and consumer, which would have been a way to optimize the example above. However, in this case, the use of the dataflow loop containing the producer and consumer requires the use of a very large PIPO buffer ($2 \times M \times N$) as shown in the following example:

```

void producer (int b[N], ...) {
    for (int j = 0; j < N; j++)
        b[f(j)] = ...;
}

void consumer(int b[N], ...) {
    for (int j = 0; j < N; j++)
        ... = b[g(j)];
}

void top(...) {
// The loop below is very constrained in terms of how it must be written
    for (int i = 0; i < M; i++) {
#pragma HLS dataflow
        int b[N];
#pragma HLS stream off variable=b

        producer(b, ...); // writes b
        consumer(b, ...); // reads b
    }
}

```

The dataflow-in-a-loop code above is also not desirable because this structure has several limitations in Vitis HLS, such as the loop structure must be very constrained (single induction variable, starting from 0 and compared with a constant or a function argument and incremented by 1).

Stream-of-Blocks Modeling Style

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

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

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

The stream-of-blocks object template is: `hls::stream_of_blocks<block_type, depth> v`

Where:

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

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

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

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

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

The main difference between `hls::stream_of_blocks` and the PIPO mechanism seen in the prior examples is that the block becomes available to the consumer as soon as the `write_lock` goes out of scope, rather than only at the return of the producer process. Hence the size of storage required to manage the prior example is much less with a stream-of-blocks than with a PIPO: namely 2N instead of 2xMxN in the example.

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



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

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

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

void top(...) {
#pragma HLS dataflow
    hls::stream_of_blocks<buf> s;

    producer(b, ...);
    consumer(b, ...);
}
```

The key features of this approach include:

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

Resource Usage

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

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



IMPORTANT! `ARRAY_RESHAPE` and `ARRAY_PARTITION` are not supported for stream-of-blocks.

Checking for Full and Empty Blocks

The `read_lock` and `write_lock` are like `while(1)` loops - they keep trying to acquire the resource until they get the resource - so the code execution will stall until the lock is acquired. You can use the `empty()` and `full()` methods as shown in the following example to determine if a call to `read_lock` or `write_lock` will stall due to the lack of available blocks to be acquired.

```
#include "hls_streamofblocks.h"

void reader(hls::stream_of_blocks<buf> &in1, hls::stream_of_blocks<buf>
&in2, int out[M][N], int c) {
    for(unsigned j = 0; j < M;) {
        if (!in1.empty()) {
            hls::read_lock<ppbuf> arr1(in1);
            for(unsigned i = 0; i < N; ++i) {
                out[j][i] = arr1[N-1-i];
            }
            j++;
        } else if (!in2.empty()) {
            hls::read_lock<ppbuf> arr2(in2);
            for(unsigned i = 0; i < N; ++i) {
                out[j][i] = arr2[N-1-i];
            }
            j++;
        }
    }
}

void writer(hls::stream_of_blocks<buf> &out1, hls::stream_of_blocks<buf>
&out2, int in[M][N], int d) {
    for(unsigned j = 0; j < M; ++j) {
        if (d < 2) {
            if (!out1.full()) {
                hls::write_lock<ppbuf> arr(out1);
                for(unsigned i = 0; i < N; ++i) {
                    arr[N-1-i] = in[j][i];
                }
            }
        } else {
            if (!out2.full()) {
                hls::write_lock<ppbuf> arr(out2);
            }
        }
    }
}
```

```

        for(unsigned i = 0; i < N; ++i) {
            arr[N-1-i] = in[j][i];
        }
    }
}

void top(int in[M][N], int out[M][N], int c, int d) {
#pragma HLS dataflow
    hls::stream_of_blocks<buf, 3> strm1, strm; // Depth=3
    writer(strm1, strm2, in, d);
    reader(strm1, strm2, out, c);
}

```

The producer and the consumer processes can perform the following actions within any scope in their body. As shown in the various examples, the scope will typically be a loop, but this is not required. Other scopes such as conditionals are also supported. Supported actions include:

- Acquire a block, i.e. an array of any supported data type.
 - In the case of the producer, the array will be empty, i.e. initialized according to the constructor (if any) of the underlying data type.
 - In the case of the consumer, the array will be full (of course in as much as the producer has filled it; the same requirements as for PIPO buffers, namely full writing if needed apply).
- Use the block for both reading and writing as if it were private local memory, up to its maximum allocated number of ports based on a BIND_STORAGE pragma or directive specified for the stream of blocks, which specifies what ports each side can see:
 - 1 port means that each side can access only one port, and the final stream-of-blocks can use a single dual-port memory for implementation.
 - 2 ports means that each side can use 1 or 2 ports depending on the schedule:
 - If the scheduler uses 2 ports on at least one side, merging will not happen;
 - If the scheduler uses 1 port, merging can happen
 - If the pragma is not specified, the scheduler will decide, based on the same criteria currently used for local arrays. Moreover:
 - The producer can both write and read the block it has acquired
 - The consumer can only read the block it has acquired
- Automatically release the block when exiting the scope in which it was acquired. A released block:
 - If released by the producer, can be acquired by the consumer.
 - If released by the consumer, can be acquired to be reused by the producer, after being re-initialized by the constructor, if any.

A stream-of-blocks is very similar in spirit to a PIPO buffer. In the case of a PIPO, acquire is the same as calling (i.e. stating) the producer or consumer process function, while the release is the same as returning from it. This means that:

- the handshakes for a PIPO are
 - ap_start/ap_ready on the consumer side and
 - ap_done/ap_continue on the producer side.
- the handshakes of a stream of blocks are
 - its own read/empty_n on the consumer side and
 - write/full_n on the producer side.

Modeling Feedback in Dataflow Regions

One main limitation of PIPO buffers is that they can flow only forward with respect to the function call sequence in C++. In other words, the following connection is not supported with PIPOs, while it can be supported with `hls::stream_of_blocks`:

```
void top(...) {
    int b[N];
    for (int i = 0; i < M; i++) {
#pragma HLS dataflow
#pragma HLS stream off variable=b
        consumer(b, ...); // reads b
        producer(b, ...); // writes b
    }
}
```

The following code example is contrived to demonstrate the concept:

```
#include "hls_streamofblocks.h"
typedef int buf[N];
void producer(hls::stream_of_blocks<buf> &out, ...) {
    for (int i = 0; i < M; i++) {
        hls::write_lock<buf> arr(out);
        for (int j = 0; j < N; j++)
            arr[f(j)] = ...;
    }
}

void consumer(hls::stream_of_blocks<buf> &in, ...) {
    if (in.empty()) // execute only when producer has already generated some
meaningful data
        return;

    for (int i = 0; i < M; i++) {
        hls::read_lock<buf> arr(in);
        for (int j = 0; j < N; j++)
            ... = arr[g(j)];
        ...
    }
}
```

```

void top(...) {
    // Note the large non-default depth.
    // The producer must complete execution before the consumer can start
    again, due to ap_ctrl_chain.
    // A smaller depth would require ap_ctrl_none
    hls::stream_of_blocks<buf, M+2> backward;

    for (int i = 0; i < M; i++) {
#pragma HLS dataflow
        consumer(backward, ...); // reads backward
        producer(backward, ...); // writes backward
    }
}

```

Limitations

There are some limitations with the use of `hls::stream_of_blocks` that you should be aware of:

- Each `hls::stream_of_blocks` object must have a single producer and consumer process, and each process must be different. In other words, local streams-of-blocks within a single process are not supported.
- You cannot use `hls::stream_of_blocks` within a sequential region. The producer and consumer must be separate concurrent processes in a dataflow region.
- You cannot use multiple nested acquire/release statements (`write_lock/read_lock`), for example in the same or nested scopes, as shown in the following example:

```

using ppbuf = int[N];
void readerImplicitNested(hls::stream_of_blocks<ppbuf>& in, ...) {
    for(unsigned j = 0; j < M; ++j) {
        hls::read_lock<ppbuf> arrA(in); // constructor would acquire A
    first
        hls::read_lock<ppbuf> arrB(in); // constructor would acquire B
    second
        for(unsigned i = 0; i < N; ++i)
            ... = arrA[f(i)] + arrB[g(i)];
        // destructor would release B first
        // destructor would release A second
    }
}

```

However, you can use multiple sequential or mutually exclusive acquire/release statements (`write_lock/read_lock`), for example inside IF/ELSE branches or in two subsequent code blocks. This is shown in the following example:

```

void readerImplicitNested(hls::stream_of_blocks<ppbuf>& in, ...) {
    for(unsigned j = 0; j < M; ++j) {
    {
        hls::read_lock<ppbuf> arrA(in); // constructor acquires A
        for(unsigned i = 0; i < N; ++i)
            ... = arrA[f(i)];
        // destructor releases A
    }
    {
        hls::read_lock<ppbuf> arrB(in); // constructor acquires B
        for(unsigned i = 0; i < N; ++i)
    }
}

```

```

        ... = arrB[g(i)];
    }
}
}
}

```

- Explicit release of locks in producer and consumer processes are not recommended, as they are automatically released when they go out of scope. However, you can use these by adding `#define EXPLICIT_ACQUIRE_RELEASE` before `#include "hls_streamofblocks.h"` in your source code.

Programming Model for Multi-Port Access in HBM

HBM provides high bandwidth if arrays are split in different banks/pseudo-channels in the design. This is a common practice in partitioning an array into different memory regions in high-performance computing. The host allocates a single buffer, which will be spread across the pseudo-channels.

Vitis HLS would consider different pointers to be independent channels, and removes any dependency analysis. But the host allocates a single buffer for both pointers, and this lets the tool maintain the dependency analysis through `pragma HLS ALIAS`. The `ALIAS` pragma informs data dependence analysis about the pointer distance. Refer to the `ALIAS` pragma for more information.

The kernel `arg0` is allocated in bank0 and kernel `arg1` is allocated in bank1. The pointer distance should be specified in the `distance` option of the `ALIAS` pragma as shown below:

```

//Assume that the host code looks like this:
int *buf = clCreateBuffer(ctx, CL_MEM_READ_ONLY, 2*bank_size, ...);
clSetKernelArg(kernel, 0, 0x20000000, buf); // bank0
clSetKernelArg(kernel, 1, 0x20000000, buf+bank_size); // bank1

//The ALIAS pragma informs data dependence analysis about the pointer
//distance
void kernel(int *bank0, int *bank1, ...)
{
#pragma HLS alias ports=bank0, bank1 distance=bank_size

```

The `ALIAS` pragma can be specified using one of the following forms:

- Constant distance:

```
#pragma HLS alias ports=arr0,arr1,arr2,arr3 distance=1024
```

- Variable distance:

```
#pragma HLS alias ports=arr0,arr1,arr2,arr3 offset=0,512,1024,2048
```

Constraints:

- The depths of all the ports in the interface pragma must be the same
- All ports must be assigned to different bundles, bound to different HBM controllers

- The number of ports specified in the second form must be the same as the number of offsets specified, one offset per port. `#pragma HLS interface offset=off` is not supported
 - Each port can only be used in one ALIAS pragma
-

Optimizing for Latency

Using Latency Constraints

Vitis HLS supports the use of a latency constraint on any scope. Latency constraints are specified using the LATENCY directive.

When a maximum and/or minimum LATENCY constraint is placed on a scope, Vitis HLS tries to ensure all operations in the function complete within the range of clock cycles specified.

The latency directive applied to a loop specifies the required latency for a single iteration of the loop: it specifies the latency for the loop body, as the following examples shows:

```
Loop_A: for ( i=0; i<N; i++) {  
#pragma HLS latency max=10  
    ..Loop Body...  
}
```

If the intention is to limit the total latency of all loop iterations, the latency directive should be applied to a region that encompasses the entire loop, as in this example:

```
Region_All_Loop_A: {  
#pragma HLS latency max=10  
Loop_A: for ( i=0; i<N; i++)  
{  
    ..Loop Body...  
}
```

In this case, even if the loop is unrolled, the latency directive sets a maximum limit on all loop operations.

If Vitis HLS cannot meet a maximum latency constraint it relaxes the latency constraint and tries to achieve the best possible result.

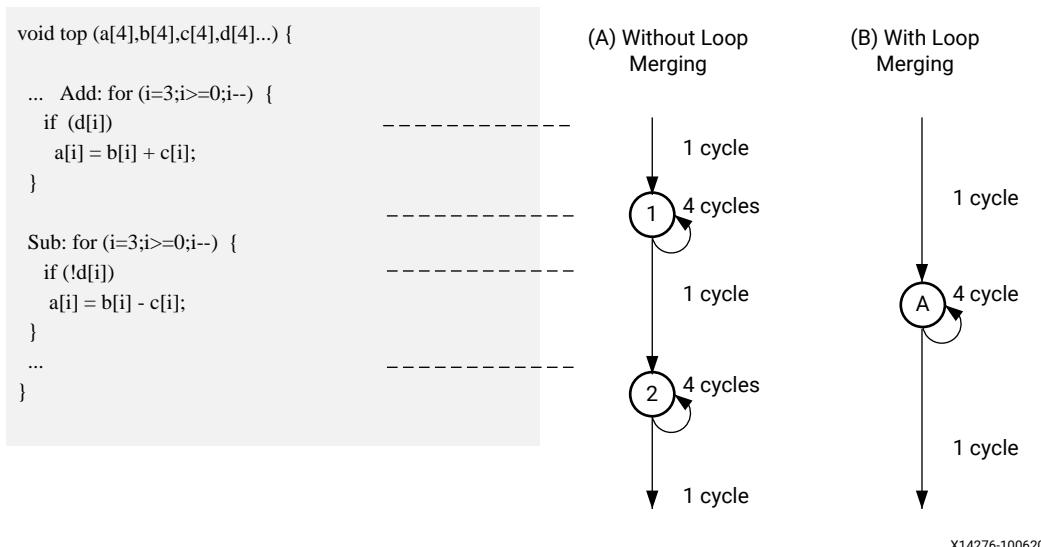
If a minimum latency constraint is set and Vitis HLS can produce a design with a lower latency than the minimum required it inserts dummy clock cycles to meet the minimum latency.

Merging Sequential Loops to Reduce Latency

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

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

Figure 96: Loop Directives



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

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

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

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

Merging loops allows the logic within the loops to be optimized together. In the example above, using a dual-port block RAM allows the add and subtraction operations to be performed in parallel.

Currently, loop merging in Vitis HLS has the following restrictions:

- If loop bounds are all variables, they must have the same value.
- If loops bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bound and constant bound cannot be merged.
- The code between loops to be merged cannot have side effects: multiple execution of this code should generate the same results ($a=b$ is allowed, $a=a+1$ is not).
- Loops cannot be merged when they contain FIFO accesses: merging would change the order of the reads and writes from a FIFO: these must always occur in sequence.

Flattening Nested Loops to Improve Latency

In a similar manner to the consecutive loops discussed in the previous section, it requires additional clock cycles to move between rolled nested loops. It requires one clock cycle to move from an outer loop to an inner loop and from an inner loop to an outer loop.

In the small example shown here, this implies 200 extra clock cycles to execute loop Outer.

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

Vitis HLS provides the `set_directive_loop_flatten` command to allow labeled perfect and semi-perfect nested loops to be flattened, removing the need to re-code for optimal hardware performance and reducing the number of cycles it takes to perform the operations in the loop.

- **Perfect loop nest:** Only the innermost loop has loop body content, there is no logic specified between the loop statements and all the loop bounds are constant.

- **Semi-perfect loop nest:** Only the innermost loop has loop body content, there is no logic specified between the loop statements but the outermost loop bound can be a variable.

For imperfect loop nests, where the inner loop has variables bounds or the loop body is not exclusively inside the inner loop, designers should try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

When the directive is applied to a set of nested loops it should be applied to the inner most loop that contains the loop body.

```
set_directive_loop_flatten top/Inner
```

Loop flattening can also be performed using the directive tab in the IDE, either by applying it to individual loops or applying it to all loops in a function by applying the directive at the function level.

Optimizing for Area

Data Types and Bit-Widths

The bit-widths of the variables in the C/C++ function directly impact the size of the storage elements and operators used in the RTL implementation. If a variables only requires 12-bits but is specified as an integer type (32-bit) it will result in larger and slower 32-bit operators being used, reducing the number of operations that can be performed in a clock cycle and potentially increasing initiation interval and latency. Refer to [Vitis HLS Memory Layout Model](#) for more information on this topic.

- Use the appropriate precision for the data types.
- Confirm the size of any arrays that are to be implemented as RAMs or registers. The area impact of any over-sized elements is wasteful in hardware resources.
- Pay special attention to multiplications, divisions, modulus or other complex arithmetic operations. If these variables are larger than they need to be, they negatively impact both area and performance.

Function Inlining

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

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

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

Inlining function `foo_sub` and using the `ALLOCATION` directive to specify only 1 instance of function `foo` is used, results in a design which only has one instance of function `foo`: one-third the area of the example above.

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

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

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

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

Array Reshaping

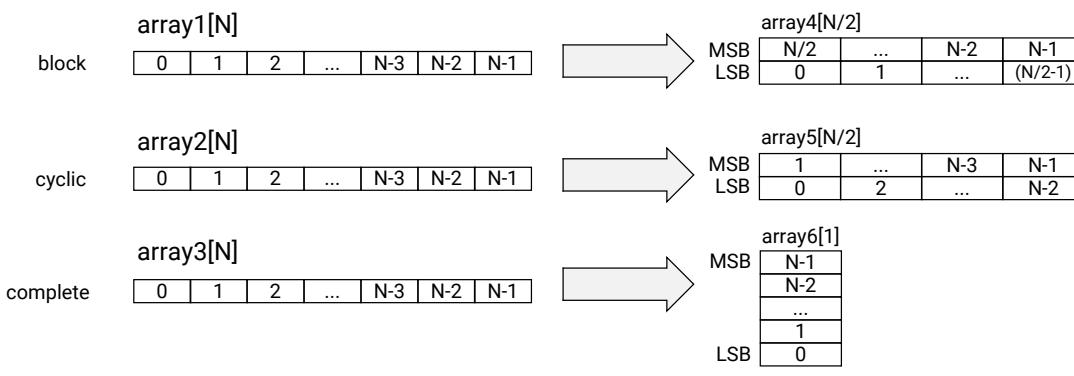
The `ARRAY_RESHAPE` directive reforms the array with a vertical mode of remapping, and is used to reduce the number of block RAM consumed while providing parallel access to the data.

Given the following example code:

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

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

Figure 97: Array Reshaping



X14307-100620

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

```
config_unroll -tripcount_threshold 16
```

Function Instantiation

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

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

Given the following code:

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

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

It is clear that function `foo` has been written to perform three exclusive operations (depending on the value of `incr`). Each instance of function `foo` is implemented in an identical manner. While this is great for function reuse and area optimization, it also means that the control logic inside the function must be more complex to account for the two exclusive operations.

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

```
void foo1() {
    // code segment 1
}

void foo2() {
    // code segment 2
}
```

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

Controlling Hardware Resources

During synthesis, Vitis HLS performs the following basic tasks:

- Elaborates the C, C++ source code into an internal database containing the operators in the C code, such as additions, multiplications, array reads, and writes.
- Maps the operators onto implementations in the hardware.

Implementations are the specific hardware components used to create the design (such as adders, multipliers, pipelined multipliers, and block RAM).

Commands, pragmas and directives provide control over each of these steps, allowing you to control the hardware implementation at a fine level of granularity.

Limiting the Number of Operators

Explicitly limiting the number of operators to reduce area may be required in some cases: the default operation of Vitis HLS is to first maximize performance. Limiting the number of operators in a design is a useful technique to reduce the area of the design: it helps reduce area by forcing the sharing of operations. However, this might cause a decline in performance.

The ALLOCATION directive allows you to limit how many operators are used in a design. For example, if a design called `foo` has 317 multiplications but the FPGA only has 256 multiplier resources (DSP macrocells). The ALLOCATION pragma shown below directs Vitis HLS to create a design with a maximum of 256 multiplication (`_mul`) operators:

```
dout_t array_arith (dio_t d[317]) {
    static int acc;
    int i;
#pragma HLS ALLOCATION instances=_mul limit=256 operation

    for (i=0;i<317;i++) {
#pragma HLS UNROLL
        acc += acc * d[i];
    }
    rerun acc;
}
```

Note: If you specify an ALLOCATION limit that is greater than needed, Vitis HLS attempts to use the number of resources specified by the limit, or the maximum necessary, which reduces the amount of sharing.

You can use the `type` option to specify if the ALLOCATION directives limits operations, implementations, or functions. The following table lists all the operations that can be controlled using the ALLOCATION directive.

Note: The operations listed below are supported by the ALLOCATION pragma or directive. The BIND_OP pragma or directive supports a subset of operators as described in the command syntax.

Table 22: Vitis HLS Operators

Operator	Description
add	Integer Addition
ashr	Arithmetic Shift-Right
dadd	Double-precision floating-point addition
dcmp	Double-precision floating-point comparison
ddiv	Double-precision floating-point division

Table 22: Vitis HLS Operators (cont'd)

Operator	Description
dmul	Double-precision floating-point multiplication
drecip	Double-precision floating-point reciprocal
drem	Double-precision floating-point remainder
drsqr	Double-precision floating-point reciprocal square root
dsub	Double-precision floating-point subtraction
dsqrt	Double-precision floating-point square root
fadd	Single-precision floating-point addition
fcmp	Single-precision floating-point comparison
fdiv	Single-precision floating-point division
fmul	Single-precision floating-point multiplication
frecip	Single-precision floating-point reciprocal
frem	Single-precision floating point remainder
frsqrt	Single-precision floating-point reciprocal square root
fsub	Single-precision floating-point subtraction
fsqrt	Single-precision floating-point square root
icmp	Integer Compare
lshr	Logical Shift-Right
mul	Multiplication
sdiv	Signed Divider
shl	Shift-Left
srem	Signed Remainder
sub	Subtraction
udiv	Unsigned Division
urem	Unsigned Remainder

Controlling Hardware Implementation

When synthesis is performed, Vitis HLS uses the timing constraints specified by the clock, the delays specified by the target device together with any directives specified by you, to determine which hardware implementations to use for various operators in the code. For example, to implement a multiplier operation, Vitis HLS could use the combinational multiplier or use a pipeline multiplier.

The implementations which are mapped to operators during synthesis can be limited by specifying the ALLOCATION pragma or directive, in the same manner as the operators. Instead of limiting the total number of multiplication operations, you can choose to limit the number of combinational multipliers, forcing any remaining multiplications to be performed using pipelined multipliers (or vice versa).

The BIND_OP or BIND_STORAGE pragmas or directives are used to explicitly specify which implementations to use for specific operations or storage types. The following command informs Vitis HLS to use a two-stage pipelined multiplier using fabric logic for variable `c`. It is left to Vitis HLS which implementation to use for variable `d`.

```
int foo (int a, int b) {
    int c, d;
    #pragma HLS BIND_OP variable=c op=mul impl=fabric latency=2
    c = a*b;
    d = a*c;

    return d;
}
```

In the following example, the BIND_OP pragma specifies that the add operation for variable `temp` is implemented using the `dsp` implementation. This ensures that the operation is implemented using a DSP module primitive in the final design. By default, add operations are implemented using LUTs.

```
void apint_arith(dinA_t  inA, dinB_t  inB,
                  dout1_t *out1
                 ) {

    dout2_t temp;
    #pragma HLS BIND_OP variable=temp op=add impl=dsp

    temp = inB + inA;
    *out1 = temp;

}
```

Refer to the BIND_OP or BIND_STORAGE pragmas or directives to obtain details on the implementations available for assignment to operations or storage types.

In the following example, the BIND_OP pragma specifies the multiplication for `out1` is implemented with a 3-stage pipelined multiplier.

```
void foo(...){
    #pragma HLS BIND_OP variable=out1 op=mul latency=3

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

}
```

If the assignment specifies multiple identical operators, the code must be modified to ensure there is a single variable for each operator to be controlled. For example, in the following code, if only the first multiplication (`inA * inB`) is to be implemented with a pipelined multiplier:

```
*out1 = inA * inB * inC;
```

The code should be changed to the following with the pragma specified on the `Result_tmp` variable:

```
#pragma HLS BIND_OP variable=Result_tmp op=mul latency=3
Result_tmp = inA * inB;
*out1 = Result_tmp * inC;
```

Optimizing Logic

Inferring Shift Registers

Vitis HLS will now infer a shift register when encountering the following code:

```
int A[N]; // This will be replaced by a shift register
for(...) {
    // The loop below is the shift operation
    for (int i = 0; i < N-1; ++i)
        A[i] = A[i+1];
    A[N] = ...;

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

Shift registers can perform a shift/cycle, which offers a significant performance improvement, and also allows a random read access per cycle anywhere in the shift register, thus it is more flexible than a FIFO.

Controlling Operator Pipelining

Vitis HLS automatically determines the level of pipelining to use for internal operations. You can use the `BIND_OP` or `BIND_STORAGE` pragmas with the `-latency` option to explicitly specify the number of pipeline stages and override the number determined by Vitis HLS.

RTL synthesis might use the additional pipeline registers to help improve timing issues that might result after place and route. Registers added to the output of the operation typically help improve timing in the output datapath. Registers added to the input of the operation typically help improve timing in both the input datapath and the control logic from the FSM.

You can use the `config_op` command to pipeline all instances of a specific operation used in the design that have the same pipeline depth. Refer to [config_op](#) for more information.

Optimizing Logic Expressions

During synthesis several optimizations, such as strength reduction and bit-width minimization are performed. Included in the list of automatic optimizations is expression balancing.

Expression balancing rearranges operators to construct a balanced tree and reduce latency.

- For integer operations expression balancing is on by default but may be disabled using the `EXPRESSION_BALANCE` pragma or directive.
- For floating-point operations, expression balancing is off by default but may be enabled.

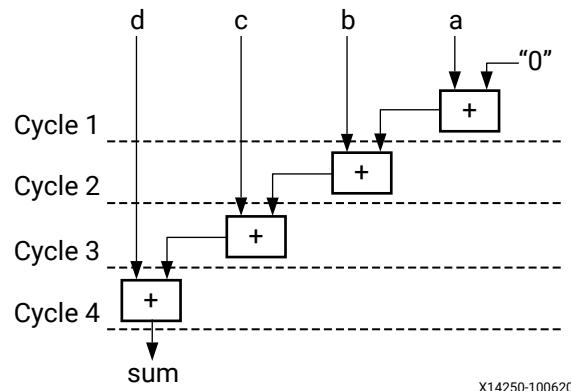
Given the highly sequential code using assignment operators such as `+=` and `*=` in the following example:

```
data_t foo_top (data_t a, data_t b, data_t c, data_t d)
{
    data_t sum;

    sum = 0;
    sum += a;
    sum += b;
    sum += c;
    sum += d;
    return sum;
}
```

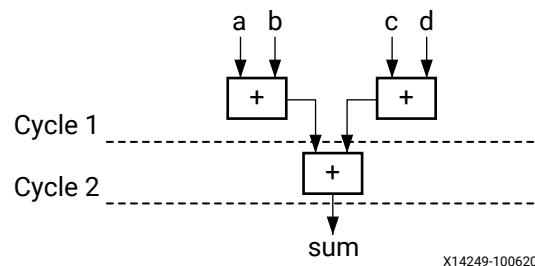
Without expression balancing, and assuming each addition requires one clock cycle, the complete computation for `sum` requires four clock cycles shown in the following figure.

Figure 98: Adder Tree



However additions `a+b` and `c+d` can be executed in parallel allowing the latency to be reduced. After balancing the computation completes in two clock cycles as shown in the following figure. Expression balancing prohibits sharing and results in increased area.

Figure 99: Adder Tree After Balancing



X14249-100620

For integers, you can disable expression balancing using the `EXPRESSION_BALANCE` optimization directive with the `off` option. By default, Vitis HLS does not perform the `EXPRESSION_BALANCE` optimization for operations of type `float` or `double`. When synthesizing `float` and `double` types, Vitis HLS maintains the order of operations performed in the C/C++ code to ensure that the results are the same as the C/C++ simulation. For example, in the following code example, all variables are of type `float` or `double`. The values of `O1` and `O2` are not the same even though they appear to perform the same basic calculation.

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

This behavior is a function of the saturation and rounding in the C/C++ standard when performing operation with types `float` or `double`. Therefore, Vitis HLS always maintains the exact order of operations when variables of type `float` or `double` are present and does not perform expression balancing by default.

You can enable expression balancing for specific operations, or you can configure the tool to enable expression balancing with `float` and `double` types using the `config_compile --unsafe_math_optimizations` command as follows:

1. In the Vitis HLS IDE, select **Solution** → **Solution Settings**.
2. In the Solution Settings dialog box, click the **General** category, select **config_compile**, and enable **unsafe_math_optimizations**.

With this setting enabled, Vitis HLS might change the order of operations to produce a more optimal design. However, the results of C/RTL co-simulation might differ from the C/C++ simulation.

The `unsafe_math_optimizations` feature also enables the `no_signed_zeros` optimization. The `no_signed_zeros` optimization ensures that the following expressions used with `float` and `double` types are identical:

```
x - 0.0 = x;
x + 0.0 = x;
0.0 - x = -x;
x - x = 0.0;
x*0.0 = 0.0;
```

Without the `no_signed_zeros` optimization the expressions above would not be equivalent due to rounding. The optimization may be optionally used without expression balancing by selecting only this option in the `config_compile` command.



TIP: When the `unsafe_math_optimizations` and `no_signed_zero` optimizations are used, the RTL implementation will have different results than the C/C++ simulation. The test bench should be capable of ignoring minor differences in the result: check for a range, do not perform an exact comparison.

Optimizing AXI System Performance

Introduction

A Vitis accelerated system includes a global memory subsystem that is used to share data between the kernels and the host application. Global memory available on the host system, outside of the Xilinx device, provides very large amounts of storage space but at the cost of longer access time compared to local memory on the Xilinx device. One of the measurements of the performance of a system/application is throughput, which is defined as the number of bytes transferred in a given time frame. Therefore, inefficient data transfers from/to the global memory will have a long memory access time which can adversely affect system performance and kernel execution time.

Development of accelerated applications in Vitis HLS should include two phases: kernel development, and improving system performance. [Design Principles for Software Programmers](#) suggested a kernel development approach implementing a cache-like Load-Compute-Store structure where the load-store functions read/write data to the global memory. Improving system performance involves implementing an efficient load and store design that can improve the kernel execution time. This chapter describes the features and metrics that can impact and improve the throughput of the load-store (LS) functions.

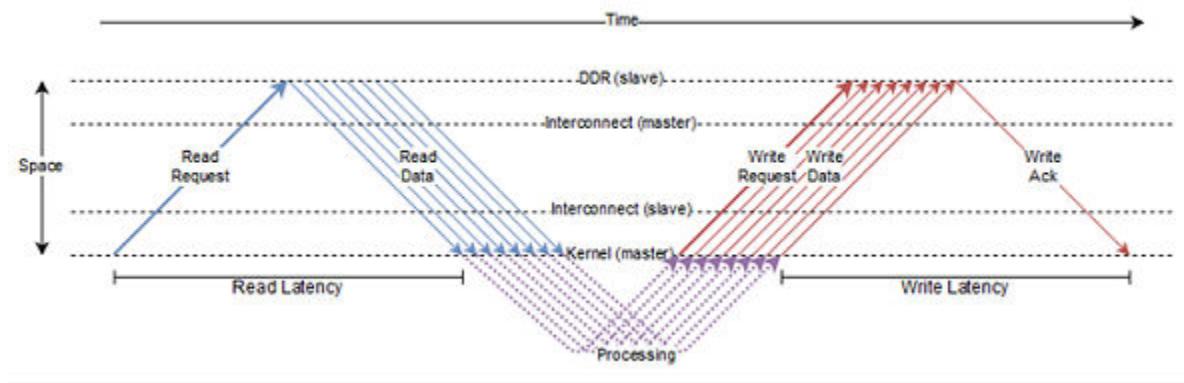
AXI Burst Transfers

Overview of Burst Transfers

Bursting is an optimization that tries to intelligently aggregate your memory accesses to the DDR to maximize the throughput bandwidth and/or minimize the latency. Bursting is one of many possible optimizations to the kernel. Bursting typically gives you a 4-5x improvement while other optimizations, like access widening or ensuring there are no dependencies through the DDR, can provide even bigger performance improvements. Typically, bursting is useful when you have contention on the DDR ports from multiple competing kernels.

The burst feature of the AXI4 protocol improves the throughput of the load-store functions by reading/writing chunks of data to or from the global memory in a single request. The larger the size of the data, the higher the throughput. This metric is calculated as follows ($(\# \text{of bytes transferred})^*$ (kernel frequency)/(Time)). The maximum kernel interface bitwidth is 512 bits, and if the kernel is compiled to run at 300 MHz then it can theoretically achieve $(512^* 300 \text{ Mhz})/1 \text{ sec} = \sim 17 \text{ GB/s}$ for a DDR.

Figure 100: AXI Protocol



The figure above shows how the AXI protocol works. The HLS kernel sends out a read request for a burst of length 8 and then sends a write request burst of length 8. The read latency is defined as the time taken between the sending of the read request burst to when the data from the first read request in the burst is received by the kernel. Similarly, the write latency is defined as the time taken between when data for the last write in the write burst is sent and the time the write acknowledgment is received by the kernel. Read requests are usually sent at the first available opportunity while write requests get queued until the data for each write in the burst becomes available.

To understand the underlying semantics of burst transfers consider the following code snippet:

```
for(size_t i = 0; i < size; i++) {
    out[f(i)] = in[f(i)];
}
```

Vitis HLS performs automatic burst optimization, which intelligently aggregates the memory accesses inside the loops/functions from the user code and performs read/write to the global memory of a particular size. These read/writes are converted into a read request, write request, and write response to the global memory. Depending on the memory access pattern Vitis HLS automatically inserts these read and write requests either outside the loop bound or inside the loop body. Depending on the placement of these requests, Vitis HLS defines two types of burst requests: sequential burst and pipelined burst.

Burst Semantics

For a given kernel, the HLS compiler implements the burst analysis optimization as a multi-pass optimization, but on a per function basis. Bursting is only done for a function and bursting across functions is not supported. The burst optimizations are reported in the [Synthesis Summary](#) report, and missed burst opportunities are also reported to help you improve burst optimization.

At first, the HLS compiler looks for memory accesses in the basic blocks of the function, such as memory accesses in a sequential set of statements inside the function. Assuming the preconditions of bursting are met, each burst inferred in these basic blocks is referred to as *sequential* burst. The compiler will automatically scan the basic block to build the longest sequence of accesses into a single sequential burst.

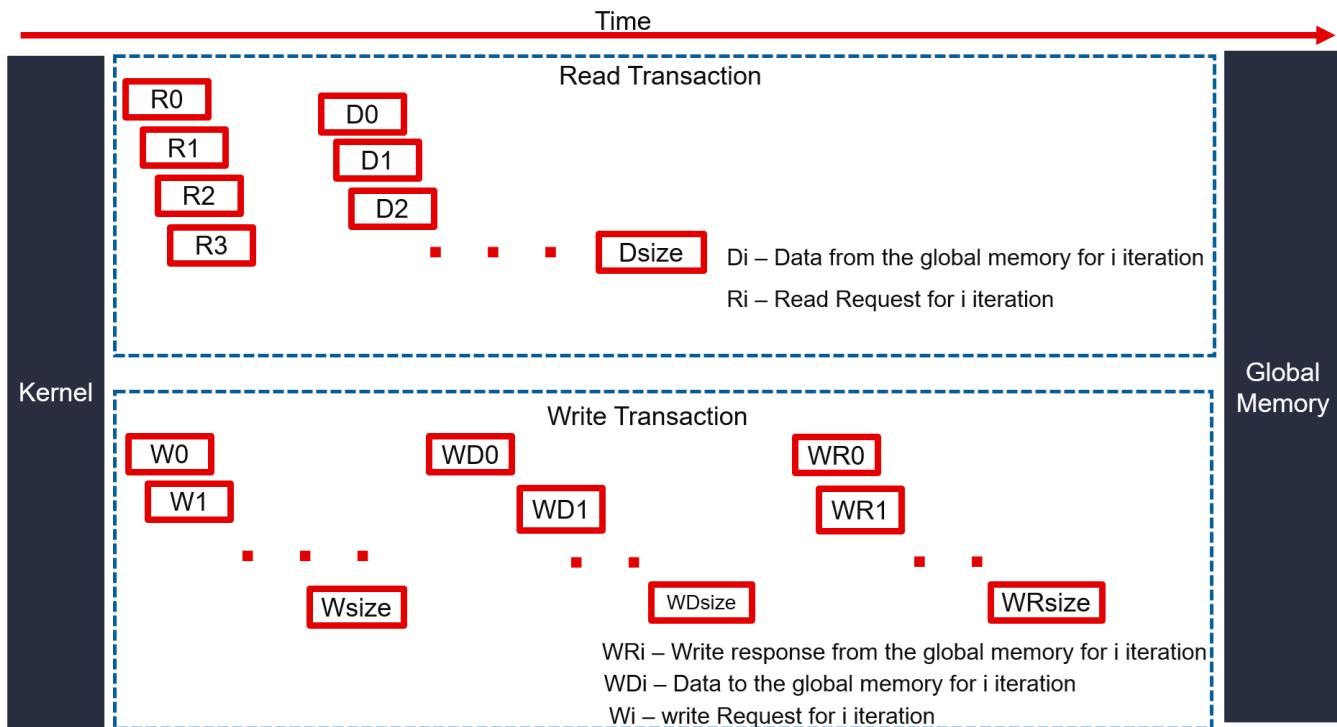
The compiler then looks at loops and tries to infer what are known as *pipeline* bursts. A pipeline burst is the sequence of reads/writes across the iterations of a loop. The compiler tries to infer the length of the burst by analyzing the loop induction variable and the trip count of the loop. If the analysis is successful, the compiler can chain the sequences of reads/writes in each iteration of the loop into one long pipeline burst. The compiler today automatically infers a pipeline or a sequential burst, but there is no way to request a specific type of burst. The code needs to be written so as to cause the tool to infer the pipeline or sequential burst.

Pipeline Burst

Pipeline burst improves the throughput of the functions by reading or writing large amounts, or the maximum amount of data in a single request. The advantage of the pipeline burst is that the future requests ($i+1$) do not have to wait for the current request (i) to finish because the read request, write request, and write response are outside the loop body and performs the requests as soon as possible, as shown in the code example below. This significantly improves the throughput of the functions as it takes less time to read/write the whole loop bound.

```
rb = ReadReq(i, size);
wb = WriteReq(i, size);
for(size_t i = 0; i < size; i++) {
    Write(wb, i) = f(Read(rb, i));
}
WriteResp(wb);
```

Figure 101: Pipeline Burst



If the compiler can successfully deduce the burst length from the induction variable (`size`) and the trip count of the loop, it will infer one big pipeline burst and will move the `ReadReq`, `WriteReq` and `WriteResp` calls outside the loop, as shown in the Pipeline Burst code example. So, the read requests for all loop iterations are combined into one read request and all the write requests are combined into one write request. Note that all read requests are typically sent out immediately while write requests are only sent out after the data becomes available.

However, if any of the preconditions of bursting are not met, as described in [Preconditions and Limitations of Burst Transfer](#), the compiler may not infer a pipeline burst but will instead try and infer a sequential burst where the `ReadReq`, `WriteReg` and `WriteResp` are alongside the read/write accesses being burst optimized, as shown in the Sequential Burst code example. In this case, the read and write requests for each loop iteration are combined into one read or write request.

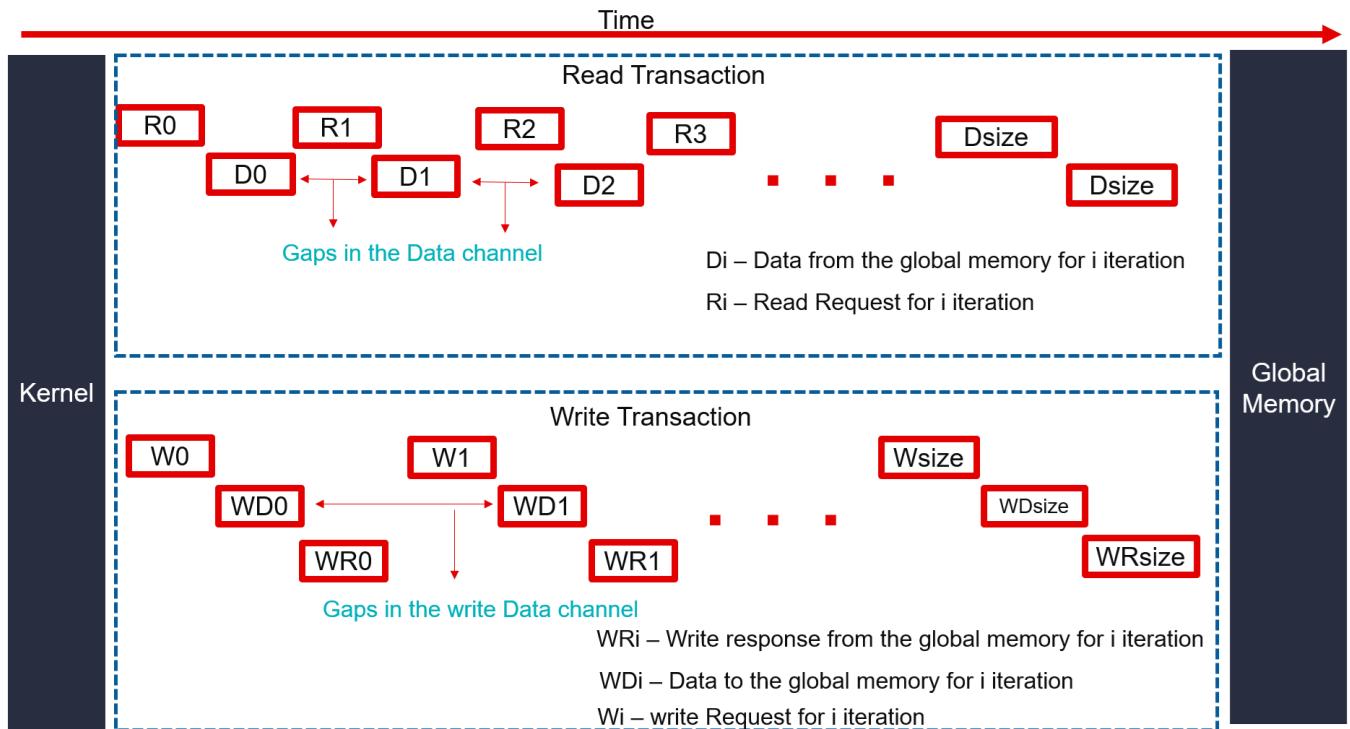
Sequential Burst

A sequential burst consists of smaller data sizes where the read requests, write requests, and write responses are inside a loop body as shown in the following code example.

```
for(size_t i = 0; i < size; i++) {
    rb = ReadReq(i, 1);
    wb = WriteReq(i, 1);
    Write(wb, i) = f(Read(rb, i));
    WriteResp(wb);
}
```

The drawback of sequential burst is that a future request ($i+1$) depends on the current request (i) finishing because it is waiting for the read request, write request, and write response to complete. This will create gaps between requests as shown in the figure below.

Figure 102: Sequential Burst



A sequential burst is not as effective as pipeline burst because it is reading or writing a small data size multiple times to compensate for the loop bounds. Although this will have a significant impact on the throughput, sequential burst is still better than no burst. Vitis HLS uses this burst technique if your code does not adhere to the [Preconditions and Limitations of Burst Transfer](#).



TIP: The size of burst requests can be further partitioned into multiple requests of user-specified size, which is controlled using the `max_read_burst_length` and `max_write_burst_length` of the INTERFACE pragma or directive, as discussed in [Options for Controlling AXI4 Burst Behavior](#).

Preconditions and Limitations of Burst Transfer

Bursting Preconditions

Bursting is about aggregating successive memory access requests. Here are the set of preconditions that these successive accesses must meet for the bursting optimization to launch successfully:

- Must be all reads, or all writes – bursting reads and writes is not possible.

- Must be a monotonically increasing order of access (both in terms of the memory location being accessed as well as in time). You cannot access a memory location that is in between two previously accessed memory locations.
- Must be consecutive in memory – one next to another with no gaps or overlap and in forward order.
- The number of read/write accesses (or burst length) must be determinable before the request is sent out. This means that even if the burst length is parametric, it must be computed before the read/write request is sent out.
- If bundling two arrays to the same M-AXI port, bursting will be done only for one array, at most, in each direction at any given time.
- There must be no dependency issues from the time a burst request is initiated and finished.

Outer Loop Burst Failure Due to Overlapping Memory Accesses

Outer loop burst inference will fail in the following example because both iteration 0 and iteration 1 of the loop L1 access the same element in arrays a and b. Burst inference is an all or nothing type of optimization - the tool will not infer a partial burst. It is a greedy algorithm that tries to maximize the length of the burst. The auto-burst inference will try to infer a burst in a bottom up fashion - from the inner loop to the outer loop, and will stop when one of the preconditions is not met. In the example below the burst inference will stop when it sees that element 8 is being read again, and so an inner loop burst of length 9 will be inferred in this case.

```
L1: for (int i = 0; i < 8; ++i)
    L2: for (int j = 0; j < 9; ++j)
        b[i*8 + j] = a[i*8 + j];

itr 0: | 0 1 2 3 4 5 6 7 8 |
itr 1: | 8 9 10 11 12 13 14 15 16 |
```

Usage of ap_int/ap_uint Types as Loop Induction Variables

Because the burst inference depends on the loop induction variable and the trip count, using non-native types can hinder the optimization from firing. It is recommended to always use unsigned integer type for the loop induction variable.

Must Enter Loop at Least Once

In some cases, the compiler can fail to infer that the max value of the loop induction variable can never be zero – that is, if it cannot prove that the loop will always be entered. In such cases, an assert statement will help the compiler infer this.

```
assert (N > 0);
L1: for(int a = 0; a < N; ++a) { ... }
```

Inter or Intra Loop Dependencies on Arrays

If you write to an array location and then read from it in the same iteration or the next, this type of array dependency can be hard for the optimization to decipher. Basically, the optimization will fail for these cases because it cannot guarantee that the write will happen before the read.

Conditional Access to Memory

If the memory accesses are being made conditionally, it can cause the burst inferencing algorithm to fail as it cannot reason through the conditional statements. In some cases, the compiler will simplify the conditional and even remove it but it is generally recommended to not use conditional statements around the memory accesses.

M-AXI Accesses Made from Inside a Function Called from a Loop

Cross-functional array access analysis is not a strong suit for compiler transformations such as burst inferencing. In such cases, users can inline the function using the `INLINE` pragma or directive to avoid burst failures.

```
void my_function(hls::stream<T> &out_pkt, int *din, int input_idx) {
    T v;
    v.data = din[input_idx];
    out_pkt.write(v);
}

void my_kernel(hls::stream<T> &out_pkt,
              int             *din,
              int             num_512_bytes,
              int             num_times) {
#pragma HLS INTERFACE mode=m_axi port = din offset=slave bundle=gmem0
#pragma HLS INTERFACE mode=axis port=out_pkt
#pragma HLS INTERFACE mode=s_axilite port=din bundle=control
#pragma HLS INTERFACE mode=s_axilite port=num_512_bytes bundle=control
#pragma HLS INTERFACE mode=s_axilite port=num_times bundle=control
#pragma HLS INTERFACE mode=s_axilite port=return bundle=control

unsigned int idx = 0;
L0: for (int i = 0; i < ntentimes; ++i) {
    L1: for (int j = 0; j < num_512_bytes; ++j) {
#pragma HLS PIPELINE
        my_function(out_pkt, din, idx++);
    }
}
```

Burst inferencing will fail because the memory accesses are being made from a called function. For the burst inferencing to work, it is recommended that users inline any such functions that are making accesses to the M-AXI memory.

An additional reason the burst inferencing will fail in this example is that the memory accessed through `din` in `my_function`, is defined by a variable (`idx`) which is not a function of the loop induction variables `i` and `j`, and therefore may not be sequential or monotonic. Instead of passing `idx`, use `(i*num_512_bytes+j)`.

Pipelined Burst Inference on a Dataflow Loop

Burst inference is not supported on a loop that has the DATAFLOW pragma or directive. However, each process/task inside the dataflow loop can have bursts. Also, sharing of M-AXI ports is not supported inside a dataflow region because the tasks can execute in parallel.

Options for Controlling AXI4 Burst Behavior

An optimal AXI4 interface is one in which the design never stalls while waiting to access the bus, and after bus access is granted, the bus never stalls while waiting for the design to read/write. There are many elements of the design that affect the system performance and burst transfer, such as the following:

- Latency
- Port Width
- Multiple Ports
- Specified Burst Length
- Number of Outstanding Reads/Writes

Latency

The read latency is defined as the time taken between sending the burst read request to when the kernel receives the data from the first read request in the burst. Similarly, the write latency is defined as the time it takes between when data for the last write in the burst is sent and the time the write response is received by the kernel. These latencies can be non-deterministic since they depend on system characteristics such as congestion on the DDR access. Because of this Vitis HLS can not accurately determine the memory read/write latency during synthesis, and so uses a default latency of 64 kernel cycles to schedule the requests and operations as below.

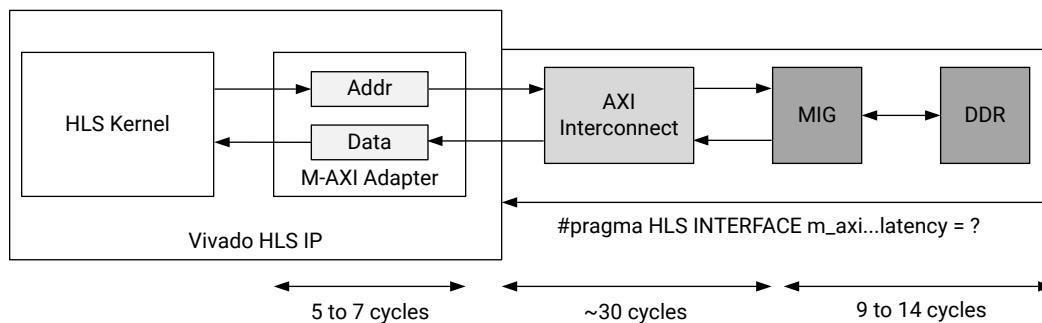
- It schedules the read/write requests and waits for the data, in parallel perform memory-independent operations, such as working on streams or compute
- Wait to schedule new read/write requests



TIP: The default tool latency can be changed using the [LATENCY](#) pragma or directive.

To help you understand the various latencies that are possible in the system, the following figure shows what happens when an HLS kernel sends a burst to the DDR.

Figure 103: Burst Transaction Diagram



X24687-100620

When your design makes a read/write request, the request is sent to the DDR through several specialized helper modules. First, the M-AXI adapter serves as a buffer for the requests created by the HLS kernel. The adapter contains logic to cut large bursts into smaller ones (which it needs to do to prevent hogging the channel or if the request crosses the 4 KB boundary, see *Vivado Design Suite: AXI Reference Guide* ([UG1037](#))), and can also stall the sending of burst requests (depending on the maximum outstanding requests parameter) so that it can safely buffer the entirety of the data for each kernel. This can slightly increase write latency but can resolve deadlock due to concurrent requests (read or write) on the memory subsystem. You can configure the M-AXI interface to hold the write request until all data is available using `config_interface -m_axi_conservative_mode`.

Getting through the adapter will cost a few cycles of latency, typically 5 to 7 cycles. Then, the request goes to the AXI interconnect that routes the kernel's request to the MIG and then eventually to the DDR. Getting through the interconnect is expensive in latency and can take around 30 cycles. Finally, getting to the DDR and back can cost anywhere from 9 to 14 cycles. These are not precise measurements of latency but rather estimates provided to show the relative latency cost of these specialized modules. For more precise measurements, you can test and observe these latencies using the Application Timeline report for your specific system, as described in [AXI Performance Case Study](#).



TIP: For information about the Application Timeline report, see [Application Timeline](#) in the Vitis Unified Software Platform Documentation.

Another way to view the latencies in the system is as follows: the interconnect has an average II of 2 while the DDR controller has an average II of 4-5 cycles on requests (while on the data they are both II=1). The interconnect arbitration strategy is based on the size of read/write requests, and so data requested with longer burst lengths get prioritized over requests with shorter bursts (thus leading to a bigger channel bandwidth being allocated to longer bursts in case of

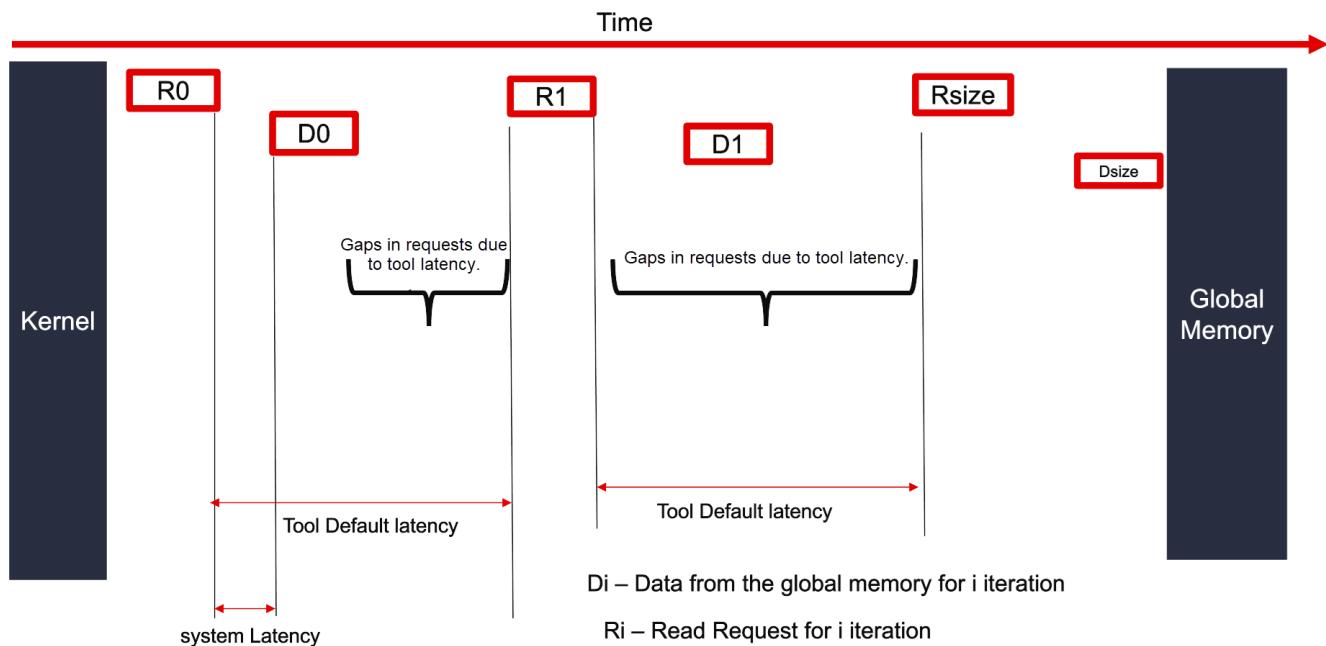
contention). Of course, a large burst request has the side-effect of preventing anyone else from accessing the DDR, and therefore there must be a compromise between burst length and reducing DDR port contention. Fortunately, the large latencies help prevent some of this port contention, and effective pipelining of the requests can significantly improve the bandwidth throughput available in the system.

Latency does not affect loops/functions with pipelined bursts since the burst requests the maximum size in a single request.

Latency effects loops/functions with sequential burst in two possible ways:

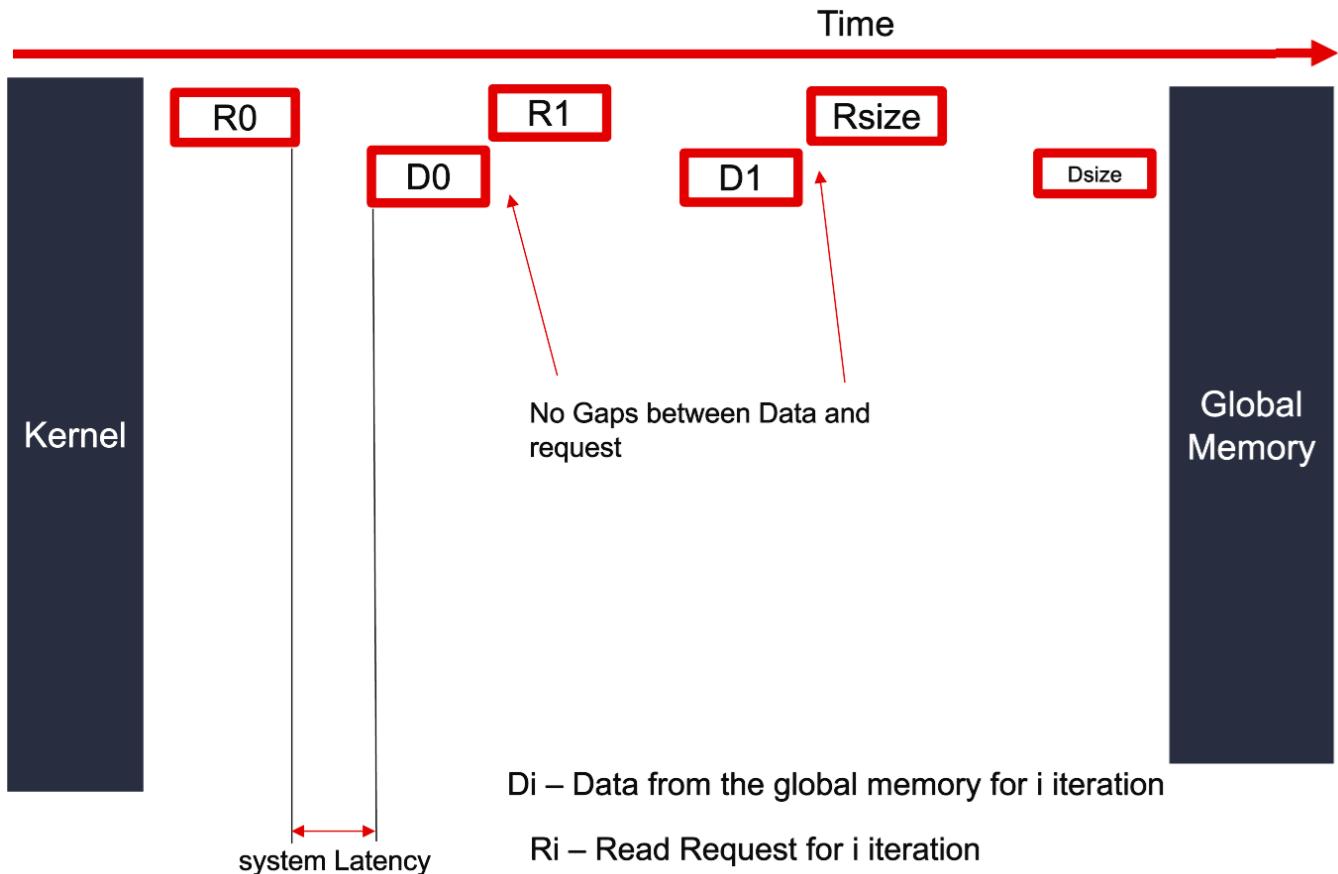
- If the system read/write latency is larger than the default tool latency, Vitis HLS has to wait for the data. Changing the LATENCY pragma or directive will not improve the performance of the system.
- If the read/write latency is less than the tool default, then Vitis HLS sits in an ideal state and wastes the remaining kernel cycles. This can impact the performance of the design because during this ideal state it does not perform tasks. As you can see from the figure below the difference between the system latency and the default latency parameter will cause the sequential requests to be delayed further in time. This causes a significant loss of throughput.

Figure 104: Default Tool Latency



However, when you reduce the tool latency using the LATENCY pragma or directive, Vitis HLS will tightly pack the requests for a sequential burst, as shown in the following figure.

Figure 105: Adjusted Tool Latency



RECOMMENDED: *Latency has a significant impact on sequential burst. Decreasing the default tool latency of the tool can improve the system performance.*

Port Width

The throughput of load-store functions can be further improved by maximizing the number of bytes transferred. Vitis HLS supports kernel ports up to 512 bits wide, which means that a kernel can read or write up to 64 bytes per clock cycle per port.

RECOMMENDED: *You should maximize the port width of the interface, i.e., the bit-width of each AXI port, by setting it to 512 bits (64 bytes).*

Vitis HLS also supports automatic port width optimization by analyzing the memory access pattern of the source code. If the code satisfies the preconditions and limitations for burst access, it will automatically resize the port to 512 bit width in the Vitis kernel flow.

IMPORTANT! *If the size and number of iterations are variable at compile time, then the tool will not automatically widen port widths.*

If the tool cannot automatically widen the port, you can manually change the port width by using [Vector Data Types](#) or [Arbitrary Precision \(AP\) Data Types](#) as the data type of the port.

Multiple Ports

The throughput of load-store functions can be further improved by maximizing concurrent read/writes. In Vitis HLS, the function arguments by default are bundled/mapped/grouped to a single port. Bundling ports into a single port helps save resources ([link to the resource impact](#)).

However, a single port can limit the performance of the kernel because all the memory transfers have to go through a single port. The `m_axi` interface has independent READ and WRITE channels, so a single port can read and write simultaneously.

Using multiple ports lets you increase the bandwidth and throughput of the kernel by creating multiple interfaces to connect to different memory banks, as shown in the [Multi-DDR tutorial](#), or the accesses will be sequential. When multiple arguments are accessing the same memory port or memory bank, an arbiter will sequence the concurrent accesses to the same memory port or bank. Having multiple ports connected to different memory banks increases the throughput of the LS functions, and as a result, the compute block should also be equally scaled to meet the throughput demand from the LS functions otherwise it will put back-pressure or stalls on the load-store functions.



RECOMMENDED: Analyze the concurrent memory reads/writes and have a dedicated/independent port for concurrent accesses.

Number of Outstanding Reads/Writes

The throughput of load-store functions can be further improved by allowing the system to hide some of the memory latency. The `m_axi_num_read_outstanding` and `m_axi_num_write_outstanding` options of the `config_interface` command, or of the INTERFACE pragma or directive, lets the Kernel control the number of pipelined memory requests sent to the global memory without waiting for the previous request to complete.

Increasing the number of pipelined requests increases the pipeline depth of the read/write requests, which will cost additional BRAM/URAM resources.

Note: In most cases where burst length ≥ 16 , the default number of outstanding reads/writes should be sufficient. For a burst of size less than 16, Xilinx recommends doubling the size of the number of outstanding from the default of 16.

Defining Burst Attributes with the INTERFACE Pragma

To create the optimal AXI4 interface, the following command options are provided in the INTERFACE directive to specify the behavior of the bursts and optimize the efficiency of the AXI4 interface.

Note that some of these options can use internal storage to buffer data and this may have an impact on area and resources:

- **latency:** Specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request several cycles (*latency*) before the read or write is expected. If this figure is too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be granted but the bus may stall waiting on the design to start the access. Default latency in Vitis HLS is 64.
- **max_read_burst_length:** Specifies the maximum number of data values read during a burst transfer. Default value is 16.
- **num_read_outstanding:** Specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design: a FIFO of size `num_read_outstanding*max_read_burst_length*word_size`. Default value is 16.
- **max_write_burst_length:** Specifies the maximum number of data values written during a burst transfer. Default value is 16.
- **num_write_outstanding:** Specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design: a FIFO of size `num_read_outstanding*max_read_burst_length*word_size`. Default value is 16.

The following INTERFACE pragma example can be used to help explain these options:

```
#pragma HLS interface mode=m_axi port=input offset=slave
bundle=gmem0
depth=1024*1024*16/(512/8) latency=100 num_read_outstanding=32
num_write_outstanding=32
max_read_burst_length=16 max_write_burst_length=16
```

- The interface is specified as having a latency of 100. The HLS compiler seeks to schedule the request for burst access 100 clock cycles before the design is ready to access the AXI4 bus.
- To further improve bus efficiency, the options `num_write_outstanding` and `num_read_outstanding` ensure the design contains enough buffering to store up to 32 read and/or write accesses. Each request will require its own buffer. This allows the design to continue processing until the bus requests are serviced.
- Finally, the options `max_read_burst_length` and `max_write_burst_length` ensure the maximum burst size is 16 and that the AXI4 interface does not hold the bus for longer than this. The HLS tool will partition longer bursts according to the specified burst length, and report this condition with a message like the following:

```
Multiple burst reads of length 192 and bit width 128 in loop
'VITIS_LOOP_2'('./src/filter.cpp:247:21) has been inferred on port
'mm_read'.
These burst requests might be further partitioned into multiple requests
during RTL generation based on the max_read_burst_length settings.
```

Commands to Configure the Burst

These commands configure global settings for the tool to optimize the AXI4 interface for the system in which it will operate. The efficiency of the operation depends on these values being set accurately. The provided default values are conservative, and may require changing depending on the memory access profile of your design.

Table 23: Vitis HLS Controls

Vitis HLS Command	Value	Description
config_rtl - m_axi_conservative_mode	bool default=false	Delay M-AXI each write request until the associated write data are entirely available (typically, buffered into the adapter or already emitted). This can slightly increase write latency but can resolve deadlock due to concurrent requests (read or write) on the memory subsystem.
config_interface - m_axi_latency	uint 0 is auto default=0 (for Vivado IP flow) default=64 (for Vitis Kernel flow)	Provide the scheduler with an expected latency for M-AXI accesses. Latency is the delay between a read request and the first read data, or between the last write data and the write response. Note that this number need not be exact, underestimation makes for a lower-latency schedule, but with longer dynamic stalls. The scheduler will account for the additional adapter latency and add a few cycles.
config_interface - m_axi_min_bitwidth	uint default=8	Minimum bitwidth for M-AXI interfaces data channels. Must be a power-of-two between 8 and 1024. Note that this does not necessarily increase throughput if the actual accesses are smaller than the required interface.
config_interface - m_axi_max_bitwidth	uint default=1024	Minimum bitwidth for M-AXI interfaces data channels. Must be a power-of-two between 8 and 1024. Note that this does decrease throughput if the actual accesses are bigger than the required interface as they will be split into a multi-cycle burst of accesses.
config_interface - m_axi_max_widen_bitwidth	uint default=0 (for Vivado IP flow) default=512 (for Vitis Kernel flow)	Allow the tool to automatically widen bursts on M-AXI interfaces up to the chosen bitwidth. Must be a power-of-two between 8 and 1024. Note that burst widening requires strong alignment properties (in addition to burst).
config_interface - m_axi_auto_max_ports	bool default=false	If the option is false, all the M-AXI interfaces that are not explicitly bundled will be bundled into a single common interface, thus minimizing resource usage (single adapter). If the option is true, all the M-AXI interfaces that are not explicitly bundled will be mapped into individual interfaces, thus increasing the resource usage (multiple adapters).

Table 23: Vitis HLS Controls (cont'd)

Vitis HLS Command	Value	Description
config_interface - m_axi_alignment_byte_size	uint default=0 (for Vivado IP flow) default=64 (for Vitis Kernel flow)	Assume top function pointers that are mapped to M-AXI interfaces are at least aligned to the provided width in byte (power of two). This can help automatic burst widening. Warning: behavior will be incorrect if the pointer are not actually aligned at runtime.
config_interface - m_axi_num_read_outstanding	uint default=16	Default value for M-AXI num_read_outstanding interface parameter.
config_interface - m_axi_num_write_outstanding	uint default=16	Default value for M-AXI num_write_outstanding interface parameter.
config_interface - m_axi_max_read_burst_length	uint default=16	Default value for M-AXI max_read_burst_length interface parameter.
config_interface - m_axi_max_write_burst_length	uint default=16	Default value for M-AXI max_write_burst_length interface parameter.

Examples of Recommended Coding Styles

As described in [Synthesis Summary](#), Vitis HLS issues a report summarizing burst activities and also identifying burst failures. If bursts of variable lengths are done, then the report will mention that bursts of variable lengths were inferred. The compiler also provides burst messages that can be found in the compiler log, `vitis_hls.log`. These messages are issued before the scheduling step.

Simple Read/Write Burst Inference

The following example is the standard way of reading and writing to the DDR and inferring a read and write burst. The Vitis HLS compiler will report the following burst inferences for the example below:

```
INFO: [HLS 214-115] Burst read of variable length and bit width 32 has been
inferred on port 'gmem'
INFO: [HLS 214-115] Burst write of variable length and bit width 32 has
been inferred on port 'gmem' (./src/vadd.cpp:75:9).
```

The code for this example follows:

```
***** BEGIN EXAMPLE *****

#define DATA_SIZE 2048
// Define internal buffer max size
#define BURSTBUFSIZE 256

//TRIPCOUNT identifiers
const unsigned int c_min = 1;
const unsigned int c_max = BURSTBUFSIZE;
const unsigned int c_chunk_sz = DATA_SIZE;
```

```

extern "C" {
void vadd(int *a, int size, int inc_value) {
    // Map pointer a to AXI4-master interface for global memory access
#pragma HLS INTERFACE mode=m_axi port=a offset=slave bundle=gmem
max_read_burst_length=256 max_write_burst_length=256
    // We also need to map a and return to a bundled axilite slave interface
#pragma HLS INTERFACE mode=s_axilite port=a bundle=control
#pragma HLS INTERFACE mode=s_axilite port=size bundle=control
#pragma HLS INTERFACE mode=s_axilite port=inc_value bundle=control
#pragma HLS INTERFACE mode=s_axilite port=return bundle=control

    int burstbuffer[BURSTBUFFERSIZE];

    // Per iteration of this loop perform BURSTBUFFERSIZE vector addition
    for (int i = 0; i < size; i += BURSTBUFFERSIZE) {
#pragma HLS LOOP_TRIPCOUNT min=c_min*c_min max=c_chunk_sz*c_chunk_sz/
(c_max*c_max)
        int chunk_size = BURSTBUFFERSIZE;
        //boundary checks
        if ((i + BURSTBUFFERSIZE) > size)
            chunk_size = size - i;

        // memcpy creates a burst access to memory
        // multiple calls of memcpy cannot be pipelined and will be
scheduled sequentially
        // memcpy requires a local buffer to store the results of the
memory transaction
        memcpy(burstbuffer, &a[i], chunk_size * sizeof(int));

        // Calculate and write results to global memory, the sequential write
in a for loop can be
        // inferred as a memory burst access
        calc_write:
        for (int j = 0; j < chunk_size; j++) {
            #pragma HLS LOOP_TRIPCOUNT min=c_size_max max=c_chunk_sz
            #pragma HLS PIPELINE II=1
            burstbuffer[j] = burstbuffer[j] + inc_value;
            a[i + j] = burstbuffer[j];
        }
    }
}
}

```

Pipelining Between Bursts

The following example will infer bursts of length N:

```

for(int x=0; x < k; ++x) {
    int off = f(x);
    for(int i = 0; i < N; ++i) {
        #pragma HLS PIPELINE II=1
        ... = gmem[off + i];
    }
}

```

But notice that the outer loop is not pipelined. This means that while there is pipelining inside bursts, there won't be any pipelining between bursts.

To remedy this you can unroll the inner loop and pipeline the outer loop to get pipelining between bursts as well. The following example will still infer bursts of length N, but now there will also be pipelining between bursts leading to higher throughput:

```
for(int x=0; x < k; ++x) {
    #pragma HLS PIPELINE II=N
    int off = f(x);
    for(int i = 0; i < N; ++i) {
        #pragma HLS UNROLL
        ... = gmem[off + i];
    }
}
```

Accessing Row Data from a Two-Dimensional Array

The following is an example of reading/writing to/from a two dimensional array. Vitis HLS infers read and write bursts and issues the following messages:

```
INFO: [HLS 214-115] Burst read of length 256 and bit width 512 has been inferred on port 'gmem' (./src/row_array_2d.cpp:43:5)
INFO: [HLS 214-115] Burst write of length 256 and bit width 512 has been inferred on port 'gmem' (./src/row_array_2d.cpp:56:5)
```

Notice that a bit width of 512 is achieved in this example. This is more efficient than the 32 bit width achieved in the simple example above. Bursting wider bit widths is another way bursts can be optimized as discussed in [Automatic Port Width Resizing](#).

The code for this example follows:

```
***** BEGIN EXAMPLE *****/
// Parameters Description:
//      NUM_ROWS:          matrix height
//      WORD_PER_ROW:      number of words in a row
//      BLOCK_SIZE:        number of words in an array
#define NUM_ROWS   64
#define WORD_PER_ROW 64
#define BLOCK_SIZE (WORD_PER_ROW*NUM_ROWS)

// Default datatype is integer
typedef int DTTYPE;
typedef hls::stream<DTTYPE> my_data_fifo;

// Read data function: reads data from global memory
void read_data(DTTYPE *inx, my_data_fifo &inFifo) {
    read_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
        read_loop_jj:
        for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
            #pragma HLS PIPELINE II=1
            inFifo << inx[WORD_PER_ROW * i + jj];
        }
    }
}

// Write data function - writes results to global memory
void write_data(DTTYPE *outx, my_data_fifo &outFifo) {
```

```

write_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
        write_loop_jj:
            for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
                #pragma HLS PIPELINE II=1
                outFifo >> outx[WORD_PER_ROW * i + jj];
            }
    }
}

// Compute function is pretty simple because this example is focused on
// efficient
// memory access pattern.
void compute(my_data_fifo &inFifo, my_data_fifo &outFifo, int alpha) {
compute_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
        compute_loop_jj:
            for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
                #pragma HLS PIPELINE II=1
                DTTYPE inTmp;
                inFifo >> inTmp;
                DTTYPE outTmp = inTmp * alpha;
                outFifo << outTmp;
            }
    }
}

extern "C" {
    void row_array_2d(DTTYPE *inx, DTTYPE *outx, int alpha) {
        // AXI master interface
        #pragma HLS INTERFACE mode=m_axi port = inx offset = slave bundle = gmem
        #pragma HLS INTERFACE mode=m_axi port = outx offset = slave bundle =
gmem
        // AXI slave interface
        #pragma HLS INTERFACE mode=s_axilite port = inx bundle = control
        #pragma HLS INTERFACE mode=s_axilite port = outx bundle = control
        #pragma HLS INTERFACE mode=s_axilite port = alpha bundle = control
        #pragma HLS INTERFACE mode=s_axilite port = return bundle = control

        my_data_fifo inFifo;
        // By default the FIFO depth is 2, user can change the depth by
using
        // #pragma HLS stream variable=inFifo depth=256
        my_data_fifo outFifo;

        // Dataflow enables task level pipelining, allowing functions and
loops to execute
        // concurrently. For more details please refer to UG902.
        #pragma HLS DATAFLOW
        // Read data from each row of 2D array
        read_data(inx, inFifo);
        // Do computation with the acquired data
        compute(inFifo, outFifo, alpha);
        // Write data to each row of 2D array
        write_data(outx, outFifo);
        return;
    }
}

```

Summary

Write code in such a way that bursting can be inferred. Ensure that none of the preconditions are violated.

Bursting does not mean that you will get all your data in one shot – it is about merging the requests together into one request, but the data will arrive sequentially, one after another.

Burst length of 16 is ideal, but even burst lengths of 8 are enough. Bigger bursts have more latency while shorter bursts can be pipelined. Do not confuse bursting with pipelining, but note that bursts can be pipelined with other bursts.

If your bursts are of fixed length, you can unroll the inner loop where bursts are inferred and pipeline the outer loop. This will achieve the same burst length, but also pipelining between the bursts to enable higher throughput.

For greater throughput, focus on widening the interface up to 512 bits rather than simply achieving longer bursts.

Bigger bursts have higher priority with the AXI interconnect. No dynamic arbitration is done inside the kernel.

You can have two `m_axi` ports connected to same DDR to model mutually exclusive access inside kernel, but the AXI interconnect outside the kernel will arbitrate competing requests.

One way to get around the out-of-order access restriction is to create your own buffer in BRAM, store the bursts in this buffer and then use this buffer to do out of order accesses. This is typically called a line buffer and is a common optimization used in video processing.

Review the Burst Optimization section of the [Synthesis Summary](#) report to learn more about burst optimizations in the design, and missed burst opportunities. If automatic burst is not occurring in your design, you may want to use the `hls::burst_maxi` data type for manual burst, as described in [Using Manual Burst](#).

Using Manual Burst

Burst transfers improve the throughput of the I/O of the kernel by reading or writing large chunks of data to the global memory. The larger the size of the burst, the higher the throughput, this metric is calculated as follows ((# of bytes transferred) * (kernel frequency)/(Time)). The maximum kernel interface bitwidth is 512 bits and if the kernel is compiled at 300 MHz, then it can theoretically achieve = (80-95% efficiency of the DDR)*(512* 300 Mhz)/1 sec = ~17-19 GB/s for a DDR. As explained, Vitis HLS performs automatic burst optimizations which intelligently aggregates the memory accesses of the loops/functions from the user code and

performs read/write of a particular size in a single burst request. However, burst transfer also has requirements that can sometimes be overburdening or difficult to meet, as discussed in [Preconditions and Limitations of Burst Transfer](#). In such cases, if you are familiar with the AXI4_m_axi protocol and understand hardware transaction modeling you can implement manual burst transfers using the `hls::burst_maxi` class as described below.

hls::burst_maxi Class

The `hls::burst_maxi` class provides a mechanism to perform read/write access to the DDR. These methods will translate the class methods usage behavior into respective AXI4 protocol and send and receive requests on the AXI4 bus signals - AW, AR, WDATA, BVALID, RDATA. These methods control the burst behavior of the HLS scheduler. The adapter, which receives the commands from the scheduler, is responsible for sending the data to the DDR. These requests will adhere to the user specified INTERFACE pragma options, such as `max_read_burst_length` and `max_write_burst_length`. The class methods should only be used in the kernel code, and not in the test bench (except for the class constructor as described below).

- Constructors:

- `burst_maxi(const burst_maxi &B) : Ptr(B.Ptr) {}`
- `burst_maxi(T *p) : Ptr(p) {}`



IMPORTANT! The HLS design and test bench must be in different files, because the constructor `burst_maxi(T *p)` is only available in C-simulation model.

- Read Methods:

- `void read_request(size_t offset, size_t len);`

This method is used to perform a read request to the `m_axi` adapter. The function returns immediately if the read request queue inside `m_axi` adapter is not full.

- `offset`: Specify the memory offset from which to read the data
- `len`: Specify the scheduler burst length. This burst length is sent to the adapter, which can then convert it to the standard AXI AMBA protocol

- `T read();`

This method is used to transfer the data from the `m_axi` adapter to the scheduler FIFO. If the data is not available, `read()` will be blocking. The `read()` method should be called `len` number of times, as specified in the `read_request()`.

- Write Methods:

- `void write_request(size_t offset, size_t len);`

This method is used to perform a write request to the `m_axi` adapter. The function returns immediately if the write request queue inside `m_axi` adapter is not full.

- `offset`: Specify the memory offset into which the data should be written
- `len`: Specify the scheduler burst length. This burst length is sent to the adapter, which can then convert it to the standard AXI AMBA protocol
- `void write(const T &val, ap_int<sizeof(T)> byteenable_mask = -1);`

This method is used to transfer data from the internal buffer of the scheduler to the `m_axi` adapter. It blocks if the internal write buffer is full. The `byteenable_mask` is used to enable the bytes in the WDATA. By default it will enable all the bytes of the transfer. The `write()` method should be called `len` number of times, as specified in the `write_request()`.

- `void write_response();`

This method blocks until all write responses are back from the global memory. This method should be called the same number of times as `write_request()`.

Using Manual Burst in HLS Design

In the HLS design, when you find that automatic burst transfers are not working as desired, and you cannot optimize the design as needed, you can implement the read and write transactions using the `hls::burst_maxi` object. In this case, you will need to modify your code to replace the original pointer argument with `burst_maxi` as a function argument. These arguments must be accessed by the explicit `read` and `write` methods of the `burst_maxi` class, as shown in the following examples.

The following shows an original code sample, which uses a pointer argument to read data from global memory.

```
void dut(int *A) {
    for (int i = 0; i < 64; i++) {
        #pragma pipeline II=1
        ... = A[i]
    }
}
```

In the modified code below, the pointer is replaced with the `hls::burst_maxi<>` class objects and methods. In the example, the HLS scheduler puts 4 requests of `len 16` from port `A` to the `m_axi` adapter. The Adapter stores them inside a FIFO and whenever the AW/AR bus is available it will send the request to the global memory. In the 64 loop iterations, the `read()` command issues a blocking call that will wait for the data to come back from the global memory. After the data becomes available the HLS scheduler will read it from the `m_axi` adapter FIFO.

```
#include "hls_burst_maxi.h"
void dut(hls::burst_maxi<int> A) {
    // Issue 4 burst requests
    A.read_request(0, 16); // request 16 elements, starting from A[0]
    A.read_request(128, 16); // request 16 elements, starting from A[128]
    A.read_request(256, 16); // request 16 elements, starting from A[256]
    A.read_request(384, 16); // request 16 elements, starting from A[384]
```

```

for (int i = 0; i < 64; i++) {
    #pragma pipeline II=1
    ... = A.read(); // Read the requested data
}
}

```

In Example 2 below, the HLS scheduler/kernel puts 2 requests from port A to the adapter, the first request of `len` 2, and the second request of `len` 1, for a total of 2 write requests. It then issues corresponding, since the total burst length is 3 write commands. The Adapter stores these requests inside a FIFO and whenever the AW, W bus is available it will send the request and data to the global memory. Finally, two `write_response` commands are used, to await response for the two `write_requests`.

```

void trf(hls::burst_maxi<int> A) {
    A.write_request(0, 2);
    A.write(x); // write A[0]
    A.write_request(10, 1);
    A.write(x, 2); // write A[1] with byte enable 0010
    A.write(x); // write A[10]
    A.write_response(); // response of write_request(0, 2)
    A.write_response(); // response of write_request(10, 1)
}

```

Using Manual Burst in C-Simulation

You can pass a regular array to the top function, and the array will be transformed to `hls::burst_maxi` automatically by the constructor.



IMPORTANT! The HLS design and test bench must be in different files, because the `burst_maxi(T *p)` constructor is only valid for use in C simulation model.

```

#include "hls_burst_maxi.h"
void dut(hls::burst_maxi<int> A);

int main() {
    int Array[1000];
    dut(Array);
    .....
}

```

Using Manual Burst to Optimize Performance

Vitis HLS characterizes two types of burst behaviors: pipeline burst, and sequential burst.

- **Pipeline Burst:** Pipeline Burst improves throughput by reading or writing the maximum amount of data in a single request. The compiler infers pipeline burst if the `read_request`, `write_request` and `write_response` calls are outside the loop, as shown in the following code example. In the below example the size is a variable that is sent from the testbench.

```

9 int buf[8192];
10 in.read_request(0, size);
11 for (int i = 0; i < size; i++) {
12     #pragma HLS PIPELINE II=1
13     buf[i] = in.read();

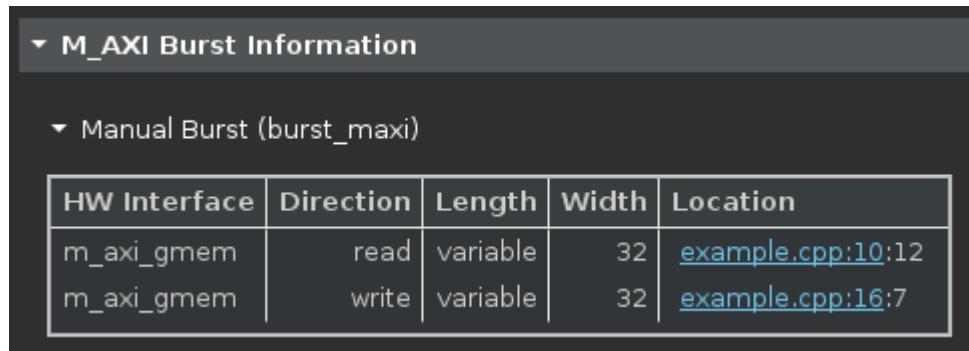
```

```

14     out.write_request(0, size*NT);
17     for (int i = 0; i < NT; i++) {
19         for (int j = 0; j < size; j++) {
20             #pragma HLS PIPELINE II=1
21             int a = buf[j];
22             out.write(a);
23     }
24 }
25 out.write_response();

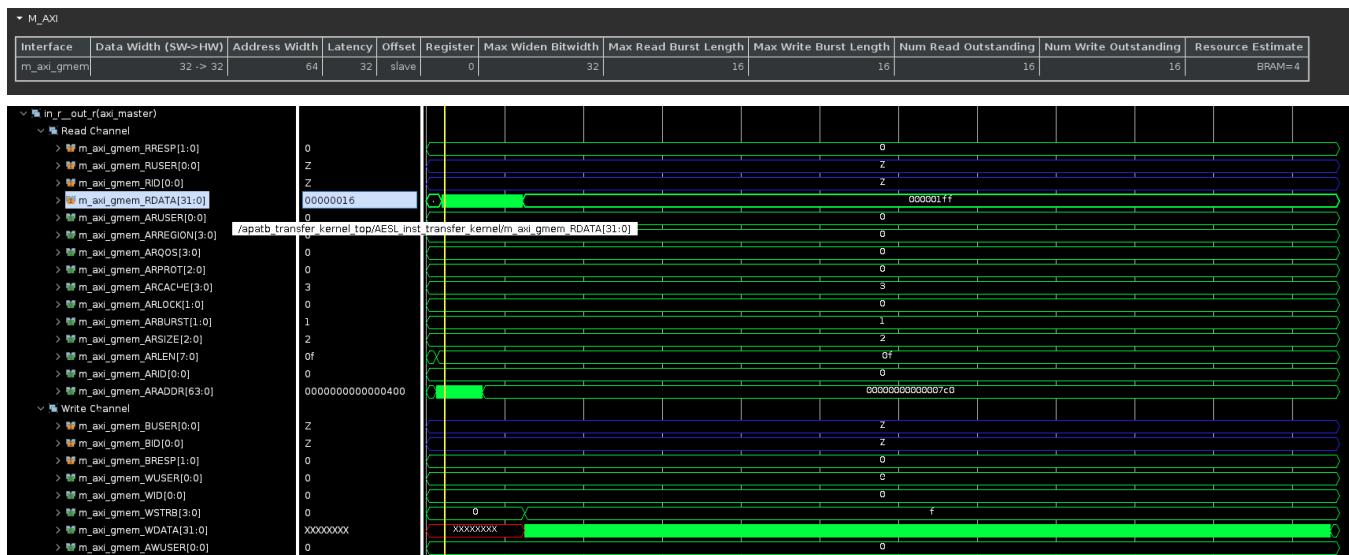
```

Figure 106: Synthesis Results



As you can see from the figure above, the tool has inferred the burst from the user code and length is mentioned as variable at compile time.

Figure 107: Performance Benefits



During the run time the HLS compiler sends a burst request of `length = size` and the adapter will partition them into the user-specified `burst_length` pragma option. In this case the default burst length is set to 16, which is used in the ARlen and AWlen channels. The read/write channel achieved maximum throughput since there are no bubbles during the transfer.

Figure 108: Co-sim Results

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
transfer_kernel				5735	5735	5735
transfer_kernel_Pipeline_VITIS_LOOP_11_1				517	517	517
transfer_kernel_Pipeline_VITIS_LOOP_18_2_VITIS_LOOP_19_3				5122	5122	5122

- **Sequential Burst:**

This burst is a sequential burst of smaller data sizes, where the read requests, write requests and write responses are inside the loop body as shown in the below snippet. The drawback of the sequential burst is that the future request ($i+1$) depends on the previous request (i) to finish since it is waiting for the read request, write request and write response to complete, this will cause gaps between requests. Sequential burst is not as effective as pipeline burst because it is reading or writing a small data size multiple times to compensate for the loop bounds. Although this will limit the improvement to throughput, sequential burst is still better than no burst.

```

void transfer_kernel(hls::burst_maxi<int> in,hls::burst_maxi<int> out,
const int size )
{
    #pragma HLS INTERFACE m_axi port=in depth=512 latency=32 offset=slave
    #pragma HLS INTERFACE m_axi port=out depth=5120 offset=slave latency=32

    int buf[8192];

    for (int i = 0; i < size; i++) {
        in.read_request(i, 1);
        #pragma HLS PIPELINE II=1
        buf[i] = in.read();
    }

    for (int i = 0; i < NT; i++) {
        for (int j = 0; j < size; j++) {
            out.write_request(j, 1);
        }
    #pragma HLS PIPELINE II=1
        int a = buf[j];
        out.write(a);
        out.write_response();
    }
}
}

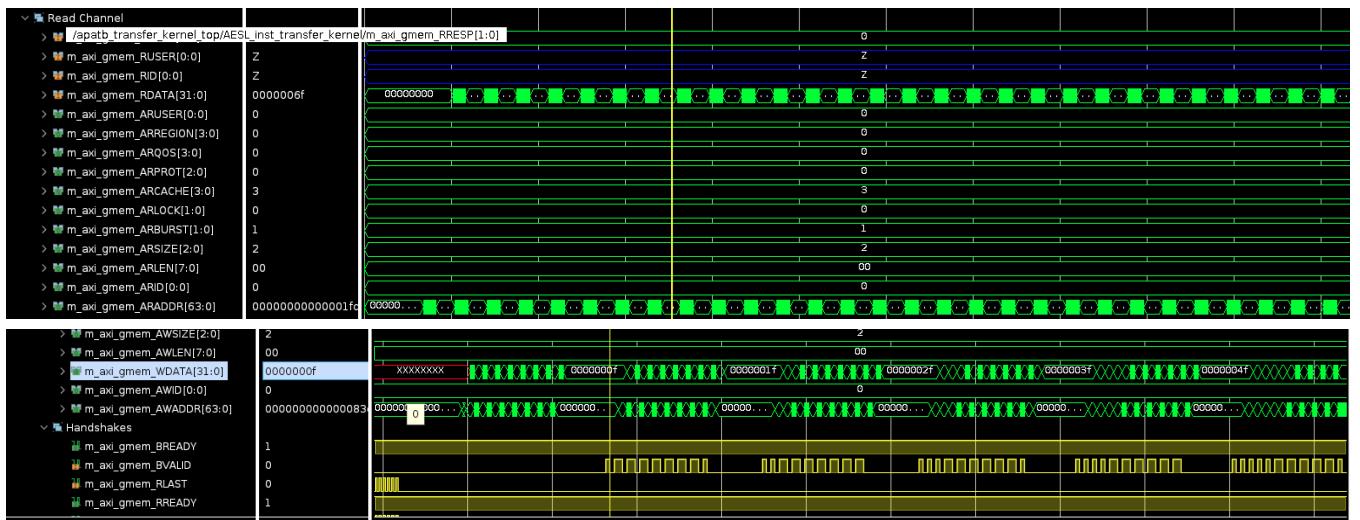
```

Figure 109: Synthesis Results

M_AXI Burst Information				
Manual Burst (burst_maxi)				
HW Interface	Direction	Length	Width	Location
m_axi_gmem	read	1	32	example.cpp:13:8
m_axi_gmem	write	1	32	example.cpp:23:15

As you can see from the report sample above, the tool achieved a burst of length =1.

Figure 110: Performance Impacts



The read/write loop R/WDATA channel has gaps equal to read/write latency, as discussed in [AXI4 Master Interface](#). For the read channel, the loop waits for all the read data to come back from the global memory. For the write channel, the innermost loop waits for the response (BVALID) to come back from the global memory. This results in performance degradation. The co-sim results also show that a 2x degradation in performance for this burst semantics.

Performance Estimates							
Max	Open Source Editor	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
transfer_kernel			13106	13106	13106		
transfer_kernel.Pipeline_VITIS_LOOP_12_1			1208	1208	1208		
transfer_kernel.Pipeline_VITIS_LOOP_20_2_VITIS_LOOP_22_3			11879	11879	11879		

Features and Limitations

1. If the `m_axi` element is a struct:

- The struct will be packed into a wide int. Disaggregation of the struct is not allowed.

- The size of struct should be a power-of-2, and should not exceed 1024 bits or the max width specified by the `config_interface -m_axi_max_bitwidth` command.
- ARRAY_PARTITION and ARRAY_RESHAPE of burst_maxi ports is not allowed.
 - You can apply the INTERFACE pragma or directive to `hls::burst_maxi`, defining an `m_axi` interface. If the `burst_maxi` port is bundled with other ports, all ports in this bundle must be `hls::burst_maxi` and must have the same element type.

```
void dut(hls::burst_maxi<int> A, hls::burst_maxi<int> B, int *C,
hls::burst_maxi<short> D) {
    #pragma HLS interface m_axi port=A offset=slave bundle=gmem // OK
    #pragma HLS interface m_axi port=B offset=slave bundle=gmem // OK
    #pragma HLS interface m_axi port=C offset=slave bundle=gmem // Bad. C
must also be hls::burst_maxi type, because it shares the same bundle
'gmem' with A and B
    #pragma HLS interface m_axi port=D offset=slave bundle=gmem // Bad. D
should have 'int' element type, because it shares the same bundle
'gmem' with A and B
}
```

- You can use the INTERFACE pragma or directive to specify the `num_read_outstanding` and `num_write_outstanding`, and the `max_read_burst_length` and `max_write_burst_length` to define the size of the internal buffer of the `m_axi` adapter.

```
void dut(hls::burst_maxi<int> A) {
    #pragma HLS interface m_axi port=A num_read_outstanding=32
    num_write_outstanding=32 max_read_burst_length=16
    max_write_burst_length=16
}
```

- The INTERFACE pragma or directive `max_widen_bitwidth` is not supported, because HLS will not change the bit width of `hls::burst_maxi` ports.
- You must make a `read_request` before `read`, or `write_request` before `write`:

```
void dut(hls::burst_maxi<int> A) {
    ... = A.read(); // Bad because read() before read_request(). You can
    catch this error in C-sim.
    A.read_request(0, 1);
}
```

- If the address and life time of the read group (`read_request()` > `read()`) and write group (`write_request()` > `write()` > `write_response()`) overlap, the tool cannot guarantee the access order. C-simulation will report an error.

```
void dut(hls::burst_maxi<int> A) {
    A.write_request(0, 1);
    A.write(x);
    A.read_request(0, 1);
    ... = A.read(); // What value is read? It is undefined. It could be
    original A[0] or updated A[0].
    A.write_response();
}

void dut(hls::burst_maxi<int> A) {
    A.write_request(0, 1);
```

```

A.write(x);
A.write_response();
A.read_request(0, 1);
... = A.read(); // this will read the updated A[0].
}

```

8. If multiple `hls::burst_maxi` ports are bundled to same `m_axi` adapter and their transaction lifetimes overlap, the behavior is unexpected.

```

void dut(hls::burst_maxi<int> A, hls::burst_maxi<int> B) {
    #pragma HLS INTERFACE m_axi port=A bundle=gmem depth = 10
    #pragma HLS INTERFACE m_axi port=B bundle=gmem depth = 10
    A.read_request(0, 10);
    B.read_request(0, 10);

    for (int i = 0; i < 10; i++) {
        #pragma HLS pipeline II=1
        .... = A.read(); // get value of A[0], A[2], A[4] ...
        .... = B.read(); // get value of A[1], A[3], A[5] ...
    }
}

```

9. Read or write requests and read or writes in different dataflow process are not supported. Dataflow checker will report an error: multiple writes in different dataflow processes are not allowed

For example:

```

void transfer(hls::burst_maxi<int> A) {
#pragma HLS dataflow
    IssueRequests(A); // issue multiple write_request() of A
    Write(A); // multiple writes to A
    GetResponse(A); // write_response() of A
}

```

Potential Pitfalls

The following are some concerns you must be aware of when implementing manual burst techniques:

- Deadlock: Improper use of manual burst can lead to deadlocks.

Too many `read_requests` before `read()` commands will cause deadlock because the `read_request` loop will push the request into the read requests FIFO, and this FIFO will only be emptied after the read from the global memory is completed. The job of the `read()` command is to read the data from the adapter FIFO and mark the request done, after which the `read_request` will be popped from the FIFO and a new request can be pushed onto it.

```

//reads/writes. will deadlock if N is larger
for (i = 0; i < N; i++)
{
    A.read_request(i * 128, 16);
    for (i = 0; i < 16 *N; i++) { ... = A.read();}

    for (int i = 0; i < N; i++) {
        p.write_request(i * 128, 16);
    }
}

```

```

for (int i = 0; i < N * 16; i++) {
    p.write(i);
}

for (int i = 0; i < N; i++) {
    p.write_response();
}

```

In the example above, if N is large then the `read_request` and read FIFO will be full as it tends to $N/2$. The read request loop would not finish, and the read command loop wouldn't start, which results in deadlock.

Note: This is case also true for `write_request()` and `write()` commands.

- AXI protocol violation: There should be an equal number of write requests and write responses. An unequal number of requests and responses would lead to AXI protocol violation

AXI Performance Case Study

Introduction

The objective of the case study is to show a step-by-step optimization to improve the throughput of the read/write loops/functions using HLS metrics. These optimizations will improve the kernel time and throughput of the system by performing efficient data transfers from global memory to the kernel. The `transfer_kernel` example below performs a DDR simple read/write (of variable size and `NUM_ITERATIONS`).



TIP: The host code, which is not shown, only transfers the data and enqueues the kernel in an in-order queue.

```

1 #include "config.h"
2 #include "assert.h"
3 extern "C" {
4     void transfer_kernel(wd* in,wd* out, const int size, const int iter )
{
5 ...
6     wd buf[256];
7     int off = (size/16);
8
9     read_loop: for (int i = 0; i < off; i++)
10    {
11        buf[i] = in[i];
12    }
13
14    write_loop: L1: for (int i = 0; i < iter; i++) {
15        L2: for (int j = 0; j < off; j++) {
16            #pragma HLS PIPELINE II=1
17            out[j+off*i] = buf[j];
18        }
19    }
20 ...
21 }
22 }
```

This case study is divided into 4 steps:

1. Baseline kernel run time with port width set to 512-bit width
2. Improve performance by changing latency parameter
3. Improve the auto burst inference of the write loop.
4. No further improvements using multiple ports and number write outstanding

Step 1: Baseline the Kernel with 512-bit Port Width

Baseline the kernel time using the default settings. During this run, the auto burst infers the following for the read and write loops:

- The Read loop achieves the pipeline burst since the tool can predict the consecutive memory access pattern. So the pipelined requests to read from the DDR of variable size.
- The Write outer loop, L1, gets sequential burst because the compiler iterates over all the combinations and identifies that since the size is unknown at compile-time, it inserts an if condition in the L1 loop before the start of the L2 loop. At the same time, the inner-most loop - L2 achieves pipeline burst. The L2 loop requests a write request of variable size, while L1 waits for all the data of L2 Loop to come back from the DDR to start the next iteration of L1.

After building and running the application, the performance can be evaluated using the Vitis Analyzer tool to view the reports generated by the build process or the run summary. Review the Burst Summary available in the Synthesis Report from Vitis HLS. It confirms the success and failures of the burst for the Read loop and Write loops.

Figure 111: Synthesis Report - Burst Summary

BURST SUMMARY					
HW Interface	Loop	Message			
m_axi_gmem	VITIS_LOOP_11_1	Multiple burst reads of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the memory controller's internal logic.			
m_axi_gmem	VITIS_LOOP_17_3	Multiple burst writes of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the memory controller's internal logic.			
BURSTS AND WIDENING MISSED					
HW Interface	Variable	Loop	Problem	Resolution	Location
m_axi_gmem	out	VITIS_LOOP_16_2	Access call is in the conditional branch	214-232	krnl_tf.cpp:16:22
m_axi_gmem	out		Could not widen since the size of type 'i512' is greater than or equal to the max_widen_bitwidth threshold of '64'.		krnl_tf.cpp:17:27
m_axi_gmem	in		Could not widen since the size of type 'i512' is greater than or equal to the max_widen_bitwidth threshold of '64'.		krnl_tf.cpp:11:26

In Vitis Analyzer, the Profile Summary and Timeline Trace reports are also useful tools to analyze the performance of the FPGA-accelerated application. In the Profile Summary the **Kernels & Compute Unit: Kernel Execution** reports the total time required by the `transfer_kernel` in the baseline build.

Figure 112: Profile Summary - Kernel Execution

Kernels & Compute Units					
Kernel Execution					
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)
transfer_kernel	300	1683.150	3.766	5.610	7.947

Step 2: Improve Performance Latency

Vitis HLS uses the default latency of 64 kernel cycles, which in some cases may not need a high value. The latency depends on the system characteristics. For this example, the latency is reduced from the default to 21 kernel cycles. The code can be changed to specify the latency using the INTERFACE pragma or directive as shown in the following example:

```

1 #include "config.h"
2 #include "assert.h"
3 extern "C" {
4     void transfer_kernel(wd* in,wd* out, const int size, const int iter )
{
5     #pragma HLS INTERFACE m_axi port=in0_index offset=slave latency=21
6     #pragma HLS INTERFACE m_axi port=out offset=slave latency=21
7 ...

```

Build and run the application and use Vitis Analyzer to review the reports generated by the build process or the run summary. Review the Synthesis Report from Vitis HLS, and examine the **HW Interface** table to see the specified latency has been applied.

Figure 113: Synthesis Report - HW Interface

M_AXI										
Interface	Data Width (SW->HW)	Address Width	Latency	Offset	Register	Max Widen Bitwidth	Max Read Burst Length	Max Write Burst Length	Num Read Outstanding	
m_axi_gmem	512 -> 512	64	21	slave	0	512	16	16		

Review the **Burst Summary** to examine the success or failures of that process.

Figure 114: Synthesis Report - Burst Summary 2

BURST SUMMARY					
HW Interface	Variable	Loop	Message	Resolution	Location
m_axi_gmem		VITIS_LOOP_11_1	Multiple burst reads of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the width of the memory interface.		
m_axi_gmem		VITIS_LOOP_17_3	Multiple burst writes of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the width of the memory interface.		
BURSTS AND WIDENING MISSED					
HW Interface	Variable	Loop	Problem	Resolution	Location
m_axi_gmem	out	VITIS_LOOP_16_2	Access call is in the conditional branch	214-232	krnl_tf.cpp:16:22
m_axi_gmem	out		Could not widen since the size of type '512' is greater than or equal to the max_widen_bitwidth threshold of '64'.		krnl_tf.cpp:17:27
m_axi_gmem	in		Could not widen since the size of type '512' is greater than or equal to the max_widen_bitwidth threshold of '64'.		krnl_tf.cpp:11:26

Examine the Kernel Execution in the Profile Summary report, and notice the performance improvement due to setting the latency for the interface.

Figure 115: Profile Summary - Kernel Execution 2

Kernels & Compute Units						
Kernel Execution						
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)	
transfer_kernel	300	1656.180	3.445	5.521	8.002	

Step 3: Improve the Automatic Burst Inference of the Write Loop

The compiler is pessimistic in auto burst inference because size and loop trip counts are unknown at compile time. You can modify the code to help the compiler infer pipelined burst, as shown below.

```

1 #include "config.h"
2 #include "assert.h"
3 extern "C" {
4     void transfer_kernel(wd* in,wd* out, const int size, const int
5     iter ) {
6         #pragma HLS INTERFACE m_axi port=in offset=slave latency=21
7         #pragma HLS INTERFACE m_axi port=out offset=slave latency=21
8
9         int k=0;
10        wd buf[256];
11        int off = (size/16);
12
13        read_loop: for (int i = 0; i <off; i++)
14        {
15            buf[i] = in[i];
16        }
17
18        write_loop: for (int j = 0; j <off*iter; j++) {
19            #pragma HLS PIPELINE II=1
20            out[k++] = buf[j%off];
21        }
22    }

```

Build and run the application and use Vitis Analyzer to review the reports generated by the build process or the run summary. The Synthesis Report confirms that the burst hints to the compiler fixed the sequential burst of the write loop. The **Burst and Widening Missed** messages are related to widening ports to 512 bits. Since this example already has a 512 port width, it can be ignored. If the width isn't 512-bit in your code, you might need to focus on resolving these messages.

Figure 116: Synthesis Report - Burst Summary 3

BURST SUMMARY			
HW Interface	Loop	Message	
m_axi_gmem	VITIS_LOOP_12_1	Multiple burst reads of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the memory access pattern.	
m_axi_gmem	VITIS_LOOP_18_2	Multiple burst writes of variable length and bit width 512. These bursts requests might be further partitioned into multiple requests during RTL generation based on the memory access pattern.	
BURSTS AND WIDENING MISSED			
HW Interface	Variable	Problem	Location
m_axi_gmem	out	Could not widen since the size of type 'i512' is greater than or equal to the max_widen_bitwidth threshold of '64'.	krnl_tf.cpp:18:27
m_axi_gmem	in	Could not widen since the size of type 'i512' is greater than or equal to the max_widen_bitwidth threshold of '64'.	krnl_tf.cpp:12:26

Examine the Kernel Execution in the Profile Summary report, and notice the performance improvement due to the latency change from Step 2, and the pipeline burst for the write loop in the current step.

Figure 117: Profile Summary - Kernel Execution 3

Kernels & Compute Units						
Kernel Execution						
Kernel	Enqueues	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)	
⚡ transfer_kernel	300	1206.800	1.748	4.023	6.720	

Summary

There are no further improvements that can be made from the Vitis HLS interface metrics. The case study example does not have concurrent read or write, so targeting multiple ports will not help in this case. In this example the tool has achieved pipeline burst for the maximum throughput, so the number of outstanding reads and writes can also be ignored. No further improvements can be confirmed from the kernel time.

As seen in the case study, implementing efficient load-store functions is dependent on the HLS interface metrics of port width, burst access, latency, multiple ports, and the number of outstanding reads and writes. Xilinx recommends the following guidelines for improving your system performance:

- Port width: Maximize the port width of the interface, i.e., the bit-width of each AXI port, by using `hls::vector` or `ap_(u)int<512>` as the data type of the port.
- Multiple ports: Analyze the concurrent memory reads/writes and have a dedicated/independent port for concurrent accesses.
- Pipeline burst: The AXI latency parameter does not have an impact on pipelined burst, the user is advised to write code to achieve the pipelined burst which can significantly improve the performance.
- Sequential burst: The AXI latency parameter has a significant impact on sequential burst, decreasing the latency number from the default latency of the tool will improve the performance.
- Num outstanding: In most of the cases of burst length $>=16$, the default num outstanding should be sufficient. For a burst of size less than 16, Xilinx recommends doubling the size of the num outstanding from the default(=16).
- Data Re-ordering: Achieving pipelined burst is always recommended, but at times because of the memory access pattern compiler can achieve only a sequential burst. In order to improve the performance, the developer can also consider different ways of storing the data in memory. For instance, accessing data in DRAM in a row-major fashion can be very inefficient. Rather than implementing a dedicated data-mover in the kernel, it may be better to transpose the data in SW and store in column-major order instead which will greatly simplify HW access patterns.

Adding RTL Blackbox Functions

The RTL blackbox enables the use of existing Verilog RTL IP in an HLS project. This lets you add RTL code to your C/C++ code for synthesis of the project by Vitis HLS. The RTL IP can be used in a sequential, pipeline, or dataflow region.



TIP: Adding an RTL blackbox to your design will restrict the tool from outputting VHDL code, Because the RTL blackbox must be Verilog, the output will be Verilog as well.

Integrating RTL IP into a Vitis HLS project requires the following files:

- C function signature for the RTL code. This can be placed into a header (.h) file.
- Blackbox JSON description file as discussed in [JSON File for RTL Blackbox](#).
- RTL IP files.

To use the RTL blackbox in an HLS project, use the following steps.

1. Call the C function signature from within your top-level function, or a sub-function in the Vitis HLS project.
2. Add the blackbox JSON description file to your HLS project using the **Add Files** command from the Vitis HLS IDE as discussed in [Creating a New Vitis HLS Project](#), or using the add_files command:

```
add_files -blackbox my_file.json
```



TIP: As explained in the next section, the new RTL Blackbox wizard can help you generate the JSON file and add the RTL IP to your project.

3. Run the Vitis HLS design flow for simulation, synthesis, and co-simulation as usual.

Requirements and Limitations

RTL IP used in the RTL blackbox feature have the following requirements:

- Should be Verilog (.v) code.
- Must have a unique clock signal, and a unique active-High reset signal.
- Must have a CE signal that is used to enable or stall the RTL IP.
- Must use the ap_ctrl_chain protocol as described in [Block-Level Control Protocols](#).

Within Vitis HLS, the RTL blackbox feature:

- Supports only C++.
- Cannot connect to top-level interface I/O signals.
- Cannot directly serve as the design-under-test (DUT).

- Does not support `struct` or `class` type interfaces.
- Supports the following interface protocols as described in [JSON File for RTL Blackbox](#):
 - **`hls::stream`**: The RTL blackbox IP supports the `hls::stream` interface. When this data type is used in the C function, use a FIFO RTL port protocol for this argument in the RTL blackbox IP.
 - **Arrays**: The RTL blackbox IP supports RAM interface for arrays. For array arguments in the C function, use one of the following RTL port protocols for the corresponding argument in the RTL blackbox IP:
 - Single port RAM – RAM_1P
 - Dual port RAM – RAM_T2P
 - **Scalars and Input Pointers**: The RTL Blackbox IP supports C scalars and input pointers only in sequential and pipeline regions. They are not supported in a dataflow region. When these constructs are used in the C function, use `wire` port protocol in the RTL IP.
 - **Inout and Output Pointers**: The RTL blackbox IP supports inout and output pointers only in sequential and pipeline regions. They are not supported in a dataflow region. When these constructs are used in the C function, the RTL IP should use `ap_vld` for output pointers, and `ap_ovld` for inout pointers.

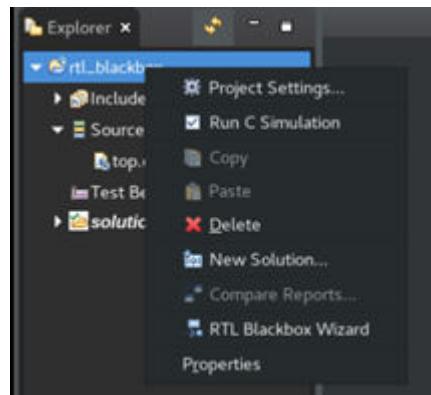


TIP: All other Vitis HLS design restrictions also apply when using RTL blackbox in your project.

Using the RTL Blackbox Wizard

Navigate to the project, right-click to open the **RTL Blackbox Wizard** as shown in the following figure:

Figure 118: Opening RTL Blackbox Wizard

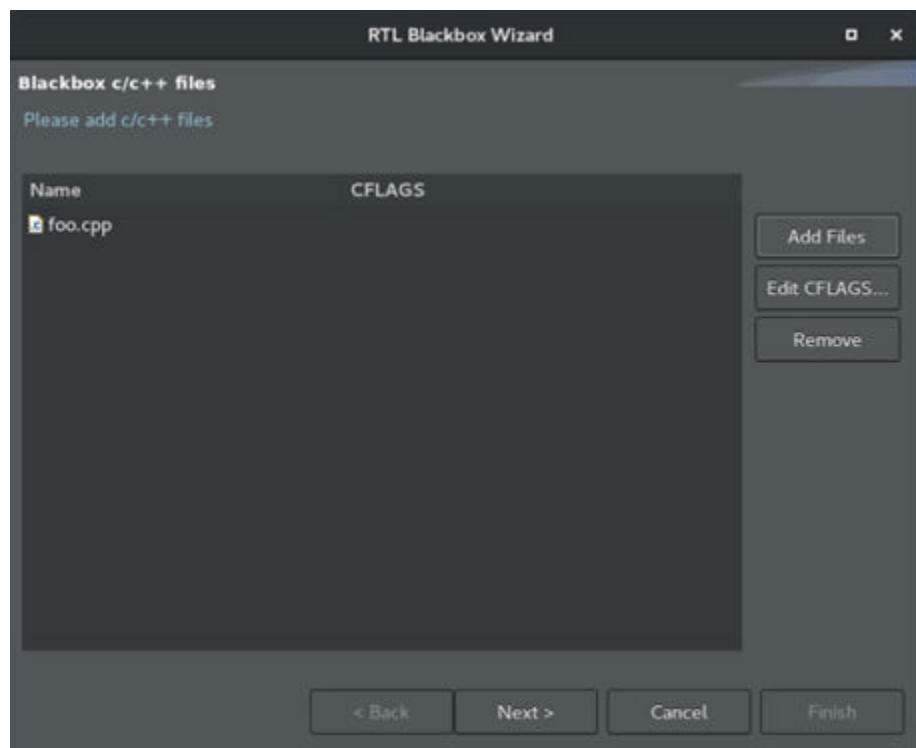


The Wizard is organized into pages that break down the process for creating a JSON file. To navigate between pages, click **Next** and select **Back**. Once the options are finalized, you can generate a JSON by clicking **OK**. Each of the following section describes each page and its input options.

C++ Model and Header Files

In the Blackbox C/C++ files page, you provide the C++ files which form the functional model of the RTL IP. This C++ model is only used during C++ simulation and C++/RTL co-simulation. The RTL IP is combined with Vitis HLS results to form the output of synthesis.

Figure 119: Blackbox C/C++ files Page



In this page, you can perform the following:

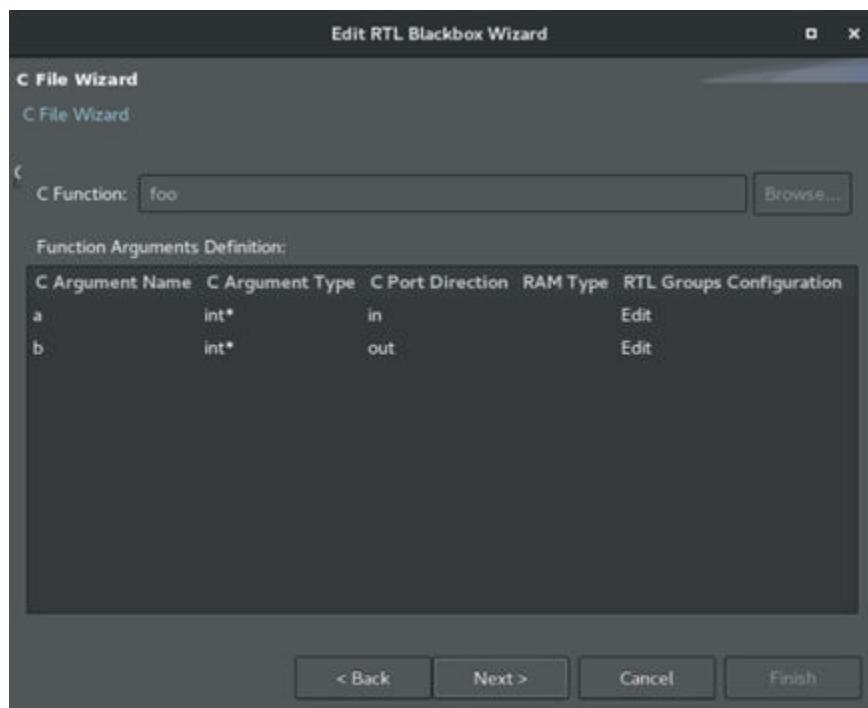
- Click **Add Files** to add files.
- Click **Edit CFLAGS** to provide a linker flag to the functional C model.
- Click **Next** to proceed.

The C File Wizard page lets you specify the values used for the C functional model of the RTL IP. The fields include:

- **C Function:** Specify the C function name of the RTL IP.

- **C Argument Name:** Specify the name(s) of the function arguments. These should relate to the ports on the IP.
- **C Argument Type:** Specify the data type used for each argument.
- **C Port Direction:** Specify the port direction of the argument, corresponding to the port in the IP.
- **RAM Type:** Specify the RAM type used at the interface.
- **RTL Group Configuration:** Specifies the corresponding RTL signal name.

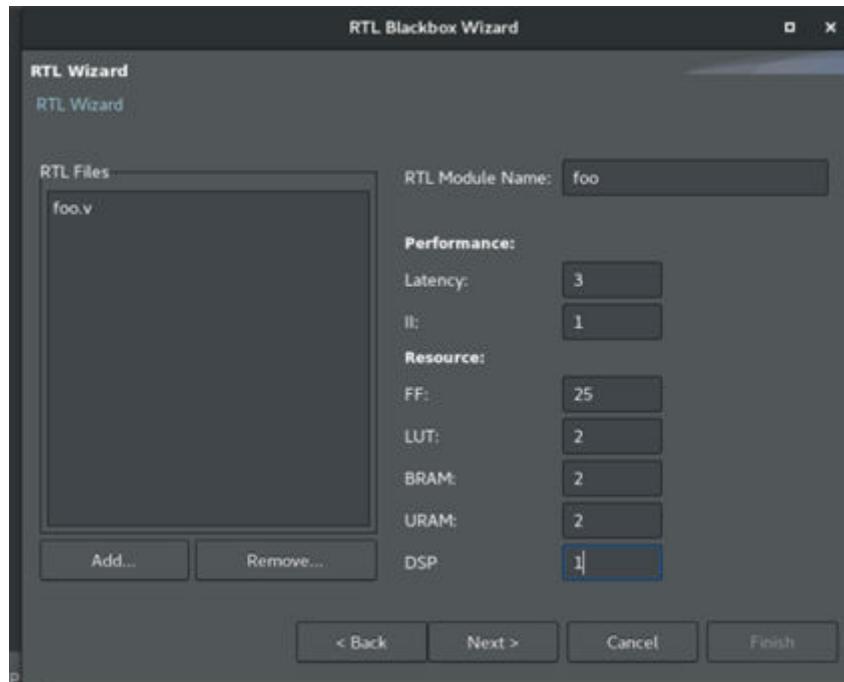
Figure 120: C File Wizard Page



Click **Next** to proceed.

RTL IP Definition

Figure 121: RTL Blackbox Wizard

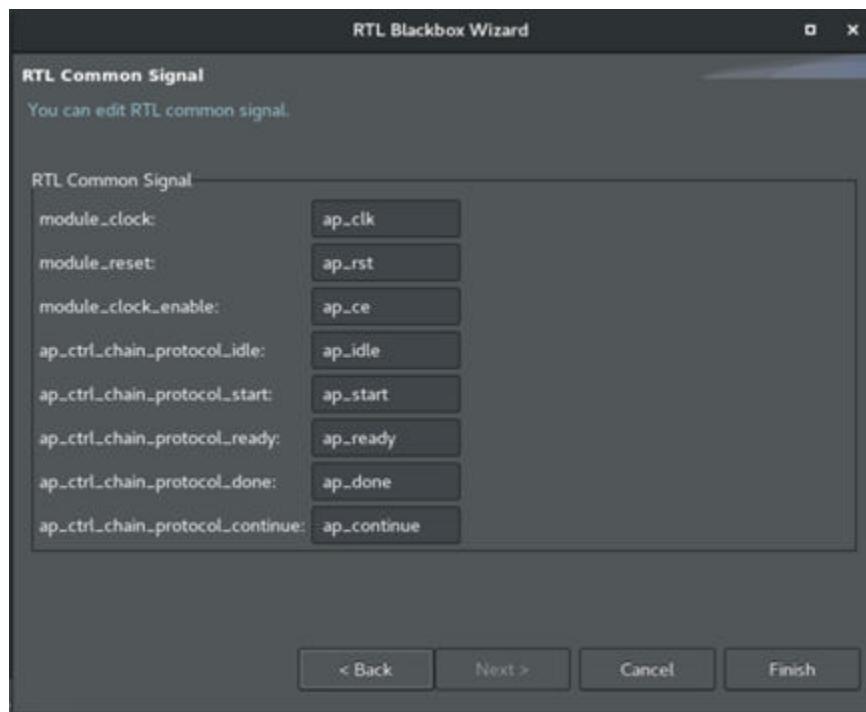


The RTL Wizard page lets you define the RTL source for the IP. The fields to define include:

- **RTL Files:** This option is used to add or remove the pre existing RTL IP files.
- **RTL Module Name:** Specify the top level RTL IP module name in this field.
- **Performance:** Specify performance targets for the IP.
 - **Latency:** Latency is the time required for the design to complete. Specify the Latency information in this field.
 - **II:** Define the target II (Initiation Interval). This is the number of clocks cycles before new input can be applied.
- **Resource:** Specify the device resource utilization for the RTL IP. The resource information provided here will be combined with utilization from synthesis to report the overall design resource utilization. You should be able to extract this information from the Vivado Design Suite

Click **Next** to proceed to the RTL Common Signal page, as shown below.

Figure 122: RTL Common Signals



- **module_clock:** Specify the name of the clock used in the RTL IP.
- **module_reset:** Specify the name of the reset signal used in the IP.
- **module_clock_enable:** Specify the name of the clock enable signal in the IP.
- **ap_ctrl_chain_protocol_start:** Specify the name of the block control start signal used in the IP.
- **ap_ctrl_chain_protocol_ready:** Specify the name of the block control ready signal used in the IP.
- **ap_ctrl_chain_protocol_done:** Specify the name of the block control done signal used in the IP.
- **ap_ctrl_chain_protocol_continue:** Specify the name of the block control continue signal used in the RTL IP.

Click **Finish** to automatically generate a JSON file for the specified IP. This can be confirmed through the log message as shown below.

Log Message:

```
" [2019-08-29 16:51:10] RTL Blackbox Wizard Information: the "foo.json" file has been created in the rtl_blackbox/Source folder."
```

The JSON file can be accessed through the Source file folder, and will be generated as described in the next section.

JSON File for RTL Blackbox

JSON File Format

The following table describes the JSON file format:

Table 24: JSON File Format

Item	Attribute	Description
c_function_name		The C++ function name for the blackbox. The <code>c_function_name</code> must be consistent with the C function simulation model.
rtl_top_module_name		The RTL function name for the blackbox. The <code>rtl_top_module_name</code> must be consistent with the <code>c_function_name</code> .
c_files	c_file	Specifies the C file used for the blackbox module.
	cflag	Provides any compile option necessary for the corresponding C file.
rtl_files		Specifies the RTL files for the blackbox module.

Table 24: JSON File Format (cont'd)

Item	Attribute	Description
c_parameters	c_name	<p>Specifies the name of the argument used for the black box C++ function.</p> <p>Unused <code>c_parameters</code> should be deleted from the template.</p>
	c_port_direction	<p>The access direction for the corresponding C argument.</p> <ul style="list-style-type: none"> <code>in</code>: Read only by blackbox C++ function. <code>out</code>: Write only by blackbox C++ function. <code>inout</code>: Will both read and write by blackbox C++ function.
	RAM_type	<p>Specifies the RAM type to use if the corresponding C argument uses the RTL RAM protocol. Two type of RAM are used:</p> <ul style="list-style-type: none"> RAM_1P: For 1 port RAM module RAM_T2P: For 2 port RAM module <p>Omit this attribute when the corresponding C argument is not using RTL 'RAM' protocol.</p>
	rtl_ports	<p>Specifies the RTL port protocol signals for the corresponding C argument (<code>c_name</code>). Every <code>c_parameter</code> should be associated with an <code>rtl_port</code>. Five type of RTL port protocols are used. Refer to the <i>RTL Port Protocols</i> table for additional details.</p> <ul style="list-style-type: none"> <code>wire</code>: An argument can be mapped to <code>wire</code> if it is a scalar or pointer with 'in' direction. <code>ap_vld</code>: An argument can be mapped to <code>ap_vld</code> if it uses pointer with 'out' direction. <code>ap_ovld</code>: An argument can be mapped to <code>ap_ovld</code> if it use a pointer with an inout direction. <code>FIFO</code>: An argument can be mapped to <code>FIFO</code> if it uses the <code>hls::stream</code> data type. <code>RAM</code>: An argument can be mapped to <code>RAM</code> if it uses an array type. The array type supports inout directions. <p>The specified RTL port protocols have associated control signals, which also need to be specified in the JSON file.</p>
c_return	c_port_direction	It must be <code>out</code> .
	rtl_ports	Specifies the corresponding RTL port name used in the RTL blackbox IP.

Table 24: JSON File Format (cont'd)

Item	Attribute	Description
rtl_common_signal	module_clock	The unique clock signal for RTL blackbox module.
	module_reset	Specifies the reset signal for RTL blackbox module. The reset signal must be active-High or positive valid.
	module_clock_enable	Specifies the clock enable signal for the RTL blackbox module. The enable signal must be active-High or positive valid.
	ap_ctrl_chain_protocol_idle	The <code>ap_idle</code> signal in the <code>ap_ctrl_chain</code> protocol for the RTL blackbox module.
	ap_ctrl_chain_protocol_start	The <code>ap_start</code> signal in the <code>ap_ctrl_chain</code> protocol for the RTL blackbox module.
	ap_ctrl_chain_protocol_ready	The <code>ap_ready</code> signal in the <code>ap_ctrl_chain</code> protocol for the RTL blackbox IP.
	ap_ctrl_chain_protocol_done	The <code>ap_done</code> signal in the <code>ap_ctrl_chain</code> protocol for blackbox RTL module.
rtl_performance	latency	Specifies the Latency of the RTL blackbox module. It must be a non-negative integer value. For Combinatorial RTL IP specify 0, otherwise specify the exact latency of the RTL module.
	II	Number of clock cycles before the function can accept new input data. It must be non-negative integer value. 0 means the blackbox can not be pipelined. Otherwise, it means the blackbox module is pipelined.
rtl_resource_usage	FF	Specifies the register utilization for the RTL blackbox module.
	LUT	Specifies the LUT utilization for the RTL blackbox module.
	BRAM	Specifies the block RAM utilization for the RTL blackbox module.
	URAM	Specifies the URAM utilization for the RTL blackbox module.
	DSP	Specifies the DSP utilization for the RTL blackbox module.

Table 25: RTL Port Protocols

RTL Port Protocol	RAM Type	C Port Direction	Attribute	User-Defined Name	Notes
wire		in	data_read_in		
ap_vld		out	data_write_out	Specifies a user defined name used in the RTL blackbox IP. As an example for wire, if the RTL port name is "flag" then the JSON FILE format is "data_read_in" : "flag".	
			data_write_valid		
ap_ovld		inout	data_read_in		
			data_write_out		
			data_write_valid		
FIFO		in	FIFO_empty_flag	Must be negative valid.	
			FIFO_read_enable		
			FIFO_data_read_in		
		out	FIFO_full_flag		
			FIFO_write_enable		
			FIFO_data_write_out		
RAM	RAM_1P	in	RAM_address		
			RAM_clock_enable		
			RAM_data_read_in		
		out	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
		inout	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
			RAM_data_read_in		

Table 25: RTL Port Protocols (cont'd)

RTL Port Protocol	RAM Type	C Port Direction	Attribute	User-Defined Name	Notes
RAM	RAM_T2P	in	RAM_address RAM_clock_enable RAM_data_read_in RAM_address_snd RAM_clock_enable_snd RAM_data_read_in_snd	Specifies a user defined name used in the RTL blackbox IP. As an example for wire, if the RTL port name is "flag" then the JSON FILE format is "data_read_in" : "flag".	Signals with _snd belong to the second port of the RAM. Signals without _snd belong to the first port.
		out	RAM_address RAM_clock_enable RAM_write_enable RAM_data_write_out RAM_address_snd RAM_clock_enable_snd RAM_write_enable_snd RAM_data_write_out_snd		
		inout	RAM_address RAM_clock_enable RAM_write_enable RAM_data_write_out RAM_data_read_in RAM_address_snd RAM_clock_enable_snd RAM_write_enable_snd RAM_data_write_out_snd RAM_data_read_in_snd		

Note: The behavioral C-function model for the RTL blackbox must also adhere to the recommended HLS coding styles.

JSON File Example

This section provides details on manually writing the JSON file required for the RTL blackbox. The following is an example of a JSON file:

```
{
  "c_function_name" : "foo",
  "rtl_top_module_name" : "foo",
  "c_files" :
  [
    {
      "c_file" : "../../a/top.cpp",
      "cflag" : ""
    }
  ]
}
```

```

        "c_file" : "xx.cpp",
        "cflag" : "-D KF"
    }
],
"rtl_files" : [
    "../foo.v",
    "xx.v"
],
"c_parameters" : [ {
    "c_name" : "a",
    "c_port_direction" : "in",
    "rtl_ports" : {
        "data_read_in" : "a"
    }
},
{
    "c_name" : "b",
    "c_port_direction" : "in",
    "rtl_ports" : {
        "data_read_in" : "b"
    }
},
{
    "c_name" : "c",
    "c_port_direction" : "out",
    "rtl_ports" : {
        "data_write_out" : "c",
        "data_write_valid" : "c_ap_vld"
    }
},
{
    "c_name" : "d",
    "c_port_direction" : "inout",
    "rtl_ports" : {
        "data_read_in" : "d_i",
        "data_write_out" : "d_o",
        "data_write_valid" : "d_o_ap_vld"
    }
},
{
    "c_name" : "e",
    "c_port_direction" : "in",
    "rtl_ports" : {
        "FIFO_empty_flag" : "e_empty_n",
        "FIFO_read_enable" : "e_read",
        "FIFO_data_read_in" : "e"
    }
},
{
    "c_name" : "f",
    "c_port_direction" : "out",
    "rtl_ports" : {
        "FIFO_full_flag" : "f_full_n",
        "FIFO_write_enable" : "f_write",
        "FIFO_data_write_out" : "f"
    }
},
{
    "c_name" : "g",
    "c_port_direction" : "in",
    "RAM_type" : "RAM_1P",
    "rtl_ports" : {
        "RAM_address" : "g_address0",

```

```

        "RAM_clock_enable" : "g_ce0",
        "RAM_data_read_in" : "g_q0"
    }
},
{
    "c_name" : "h",
    "c_port_direction" : "out",
    "RAM_type" : "RAM_1P",
    "rtl_ports" : [
        "RAM_address" : "h_address0",
        "RAM_clock_enable" : "h_ce0",
        "RAM_write_enable" : "h_we0",
        "RAM_data_write_out" : "h_d0"
    ]
},
{
    "c_name" : "i",
    "c_port_direction" : "inout",
    "RAM_type" : "RAM_1P",
    "rtl_ports" : [
        "RAM_address" : "i_address0",
        "RAM_clock_enable" : "i_ce0",
        "RAM_write_enable" : "i_we0",
        "RAM_data_write_out" : "i_d0",
        "RAM_data_read_in" : "i_q0"
    ]
},
{
    "c_name" : "j",
    "c_port_direction" : "in",
    "RAM_type" : "RAM_T2P",
    "rtl_ports" : [
        "RAM_address" : "j_address0",
        "RAM_clock_enable" : "j_ce0",
        "RAM_data_read_in" : "j_q0",
        "RAM_address_snd" : "j_address1",
        "RAM_clock_enable_snd" : "j_ce1",
        "RAM_data_read_in_snd" : "j_q1"
    ]
},
{
    "c_name" : "k",
    "c_port_direction" : "out",
    "RAM_type" : "RAM_T2P",
    "rtl_ports" : [
        "RAM_address" : "k_address0",
        "RAM_clock_enable" : "k_ce0",
        "RAM_write_enable" : "k_we0",
        "RAM_data_write_out" : "k_d0",
        "RAM_address_snd" : "k_address1",
        "RAM_clock_enable_snd" : "k_ce1",
        "RAM_write_enable_snd" : "k_we1",
        "RAM_data_write_out_snd" : "k_d1"
    ]
},
{
    "c_name" : "l",
    "c_port_direction" : "inout",
    "RAM_type" : "RAM_T2P",
    "rtl_ports" : [
        "RAM_address" : "l_address0",
        "RAM_clock_enable" : "l_ce0",
        "RAM_write_enable" : "l_we0",

```

```

        "RAM_data_write_out" : "l_d0",
        "RAM_data_read_in" : "l_q0",
        "RAM_address_snd" : "l_address1",
        "RAM_clock_enable_snd" : "l_ce1",
        "RAM_write_enable_snd" : "l_we1",
        "RAM_data_write_out_snd" : "l_d1",
        "RAM_data_read_in_snd" : "l_q1"
    }
],
"c_return" : {
    "c_port_direction" : "out",
    "rtl_ports" : {
        "data_write_out" : "ap_return"
    }
},
"rtl_common_signal" : {
    "module_clock" : "ap_clk",
    "module_reset" : "ap_rst",
    "module_clock_enable" : "ap_ce",
    "ap_ctrl_chain_protocol_idle" : "ap_idle",
    "ap_ctrl_chain_protocol_start" : "ap_start",
    "ap_ctrl_chain_protocol_ready" : "ap_ready",
    "ap_ctrl_chain_protocol_done" : "ap_done",
    "ap_ctrl_chain_protocol_continue" : "ap_continue"
},
"rtl_performance" : {
    "latency" : "6",
    "II" : "2"
},
"rtl_resource_usage" : {
    "FF" : "0",
    "LUT" : "0",
    "BRAM" : "0",
    "URAM" : "0",
    "DSP" : "0"
}
}
}

```

HLS Pragmas

Optimizations in Vitis HLS

In the Vitis software platform, a kernel defined in the C/C++ language, or OpenCL™ C, must be compiled into the register transfer level (RTL) that can be implemented into the programmable logic of a Xilinx device. The `v++` compiler calls the Vitis High-Level Synthesis (HLS) tool to synthesize the RTL code from the kernel source code.

The HLS tool is intended to work with the Vitis IDE project without interaction. However, the HLS tool also provides pragmas that can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. These pragmas can be added directly to the source code for the kernel.

The HLS pragmas include the optimization types specified in the following table.

Table 32: Vitis HLS Pragmas by Type

Type	Attributes
Kernel Optimization	<ul style="list-style-type: none">• <code>pragma HLS aggregate</code>• <code>pragma HLS bind_op</code>• <code>pragma HLS bind_storage</code>• <code>pragma HLS expression_balance</code>• <code>pragma HLS latency</code>• <code>pragma HLS reset</code>• <code>pragma HLS top</code>
Function Inlining	<ul style="list-style-type: none">• <code>pragma HLS inline</code>
Interface Synthesis	<ul style="list-style-type: none">• <code>pragma HLS interface</code>
Task-level Pipeline	<ul style="list-style-type: none">• <code>pragma HLS dataflow</code>• <code>pragma HLS stream</code>
Pipeline	<ul style="list-style-type: none">• <code>pragma HLS pipeline</code>• <code>pragma HLS occurrence</code>
Loop Unrolling	<ul style="list-style-type: none">• <code>pragma HLS unroll</code>• <code>pragma HLS dependence</code>
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>

Table 32: Vitis HLS Pragmas by Type (cont'd)

Type	Attributes
Array Optimization	<ul style="list-style-type: none"> • <code>pragma HLS array_partition</code> • <code>pragma HLS array_reshape</code>
Structure Packing	<ul style="list-style-type: none"> • <code>pragma HLS aggregate</code> • <code>pragma HLS dataflow</code>
Resource Optimization	<ul style="list-style-type: none"> • <code>pragma HLS allocation</code> • <code>pragma HLS function_instantiate</code>

pragma HLS aggregate

Description

Collects and groups the data fields of a struct into a single scalar with a wider word width.

The AGGREGATE pragma is used for grouping all the elements of a struct into a single wide vector to allow all members of the struct to be read and written to simultaneously. The bit alignment of the resulting new wide-word can be inferred from the declaration order of the struct elements. The first element takes the LSB of the vector, and the final element of the struct is aligned with the MSB of the vector.

If the struct contains arrays, the AGGREGATE pragma performs a similar operation as ARRAY_RESHAPE, and combines the reshaped array with the other elements in the struct. Any arrays declared inside the struct are completely partitioned and reshaped into a wide scalar and packed with other scalar elements.



IMPORTANT! You should exercise some caution when using the AGGREGATE optimization on struct objects with large arrays. If an array has 4096 elements of type int, this will result in a vector (and port) of width $4096 \times 32 = 131072$ bits. The Vitis HLS tool can create this RTL design, however it is very unlikely logic synthesis will be able to route this during the FPGA implementation.

Syntax

Place the pragma near the definition of the struct variable to aggregate:

```
#pragma HLS aggregate variable=<variable> compact=<arg>
```

Where:

- `variable=<variable>`: Specifies the variable to be grouped.

- `compact=[bit | byte | none | auto]`: Specifies the alignment of the aggregated struct. Alignment can be on the bit-level, the byte-level, none, or automatically determined by the tool which is the default behavior.

Example 1

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

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

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

Example 2

Aggregates struct array `AB[17]` with three 8-bit field fields (R, G, B) into a new 17 element array of 24-bits.

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

pixel AB[17];
#pragma HLS aggregate variable=AB
```

See Also

- [set_directive_aggregate](#)
- [pragma HLS array_reshape](#)
- [pragma HLS disaggregate](#)

pragma HLS alias

Description

Specify that two or more `M_AXI` pointer arguments point to the same underlying buffer in memory (DDR or HBM) and indicate any aliasing between the pointers by setting the distance or offset between them.



IMPORTANT! The ALIAS pragma applies to top-level function arguments mapped to `M_AXI` interfaces.

Vitis HLS considers different pointers to be independent channels and generally does not provide any dependency analysis. However, in cases where the host allocates a single buffer for multiple pointers, this relationship can be communicated through the ALIAS pragma or directive and dependency analysis can be maintained. The ALIAS pragma enables data dependence analysis in Vitis HLS by defining the distance between pointers in the buffer.

Requirements for ALIAS:

- All ports assigned to an ALIAS pragma must be assigned to M_AXI interfaces and assigned to different bundles, as shown in the example below
- Each port can only be used in one ALIAS pragma or directive
- The depth of all ports assigned to an ALIAS pragma must be the same
- When offset is specified, the number of ports and number of offsets specified must be the same: one offset per port
- The offset for the INTERFACE must be specified as slave or direct, offset=off is not supported

Syntax

```
pragma HLS alias ports=<list> [distance=<int> | offset=<list...>]
```

Where:

- **ports=<list>**: specifies the ports to alias.
- **distance=<integer>**: Specifies the difference between the pointer values passed to the ports in the list.
- **offset=<list>**: Specifies the offset of the pointer passed to each port in the ports list with respect to the origin of the array.

Note: offset and distance are mutually exclusive.

Example

For the following function top:

```
void top(int *arr0, int *arr1, int *arr2, int *arr3, ...) {
    #pragma HLS interface mode=m_axi port=arr0 bundle=hbm0 depth=0x40000000
    #pragma HLS interface mode=m_axi port=arr1 bundle=hbm1 depth=0x40000000
    #pragma HLS interface mode=m_axi port=arr2 bundle=hbm2 depth=0x40000000
    #pragma HLS interface mode=m_axi port=arr3 bundle=hbm3 depth=0x40000000
```

The following pragma defines aliasing for the specified array pointers, and defines the distance between them:

```
#pragma HLS ALIAS ports=arr0,arr1,arr2,arr3 distance=10000000
```

Alternatively, the following pragma specifies the `offset` between pointers, to accomplish the same effect:

```
#pragma HLS ALIAS ports=arr0,arr1,arr2,arr3
offset=00000000,10000000,20000000,30000000
```

See Also

- [set_directive_alias](#)
- [pragma HLS interface](#)

pragma HLS allocation

Description

Specifies restrictions to limit resource allocation in the implemented kernel. The ALLOCATION pragma or directive can limit the number of RTL instances and hardware resources used to implement specific functions, loops, or operations. The ALLOCATION pragma is specified inside the body of a function, a loop, or a region of code.

For example, if the C source has four instances of a function `foo_sub`, the ALLOCATION pragma can ensure that there is only one instance of `foo_sub` in the final RTL. All four instances of the C function are implemented using the same RTL block. This reduces resources used by the function, but negatively impacts performance by sharing those resources.

Template functions can also be specified for ALLOCATION by specifying the function pointer instead of the function name, as shown in the examples below.

The operations in the C code, such as additions, multiplications, array reads, and writes, can also be limited by the ALLOCATION pragma.

Syntax

Place the pragma inside the body of the function, loop, or region where it will apply.



IMPORTANT! The order of the arguments below is important. The `<type>` as operation or function must follow the `allocation` keyword.

```
#pragma HLS allocation <type> instances=<list>
limit=<value>
```

Where:

- `<type>`: The type is specified as one of the following:

- **function:** Specifies that the allocation applies to the functions listed in the `instances=` list. The function can be any function in the original C or C++ code that has *not* been:
 - Inlined by the pragma `HLS inline`, or the `set_directive_inline` command, or
 - Inlined automatically by the Vitis HLS tool.
- **operation:** Specifies that the allocation applies to the operations listed in the `instances=` list.
- **instances=<list>:** Specifies the names of functions from the C code, or operators. For a complete list of operations that can be limited using the ALLOCATION pragma, refer to the [config_op](#) command.
- **limit=<value>:** Optionally specifies the limit of instances to be used in the kernel.

Example 1

Given a design with multiple instances of function `foo`, this example limits the number of instances of `foo` in the RTL for the hardware kernel to two.

```
#pragma HLS allocation function instances=foo limit=2
```

Example 2

Limits the number of multiplier operations used in the implementation of the function `my_func` to one. This limit does not apply to any multipliers outside of `my_func`, or multipliers that might reside in sub-functions of `my_func`.



TIP: To limit the multipliers used in the implementation of any sub-functions, specify an allocation directive on the sub-functions or inline the sub-function into function `my_func`.

```
void my_func(data_t angle) {
    #pragma HLS allocation operation instances=mul limit=1
    ...
}
```

Example 3

The ALLOCATION pragma can also be used on template functions as shown below. The identification is generally based on the function name, but in the case of template functions it is based on the function pointer:

```
template <typename DT>
void foo(DT a, DT b){
}
// The following is valid
#pragma HLS ALLOCATION function instances=foo<DT>
...
// The following is not valid
#pragma HLS ALLOCATION function instances=foo
```

See Also

- [set_directive_allocation](#)
- [pragma HLS inline](#)

pragma HLS array_partition

Description



IMPORTANT! *Array_Partition* and *Array_Reshape* pragmas and directives are not supported for *M_AXI* Interfaces on the top-level function. Instead you can use the *hls::vector* data types as described in [Vector Data Types](#).

Partitions an array into smaller arrays or individual elements and provides the following:

- Results in RTL with multiple small memories or multiple registers instead of one large memory.
- Effectively increases the amount of read and write ports for the storage.
- Potentially improves the throughput of the design.
- Requires more memory instances or registers.

Syntax

Place the pragma in the C source within the boundaries of the function where the array variable is defined.

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

Where:

- **variable=<name>**: A required argument that specifies the array variable to be partitioned.

- `type=<type>`: Optionally specifies the partition type. The default type is `complete`. The following types are supported:
 - `cyclic`: Cyclic partitioning creates smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. For example, if `factor=3` is used:
 - Element 0 is assigned to the first new array
 - Element 1 is assigned to the second new array.
 - Element 2 is assigned to the third new array.
 - Element 3 is assigned to the first new array again.
 - `block`: Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the `factor=` argument.
 - `complete`: Complete partitioning decomposes the array into individual elements. For a one-dimensional array, this corresponds to resolving a memory into individual registers. This is the default `<type>`.
- `factor=<int>`: Specifies the number of smaller arrays that are to be created.



IMPORTANT! For complete type partitioning, the factor is not specified. For block and cyclic partitioning, the `factor=` is required.

- `dim=<int>`: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to <N>, for an array with <N> dimensions:
 - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
 - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.

Example 1

This example partitions the 13 element array, AB[13], into four arrays using block partitioning:

```
#pragma HLS array_partition variable=AB type=block factor=4
```



TIP: Because four is not an integer factor of 13:

- Three of the new arrays have three elements each
- One array has four elements (AB[9:12])

Example 2

This example partitions dimension two of the two-dimensional array, AB[6][4] into two new arrays of dimension [6][2]:

```
#pragma HLS array_partition variable=AB type=block factor=2 dim=2
```

Example 3

This example partitions the second dimension of the two-dimensional `in_local` array into individual elements.

```
int in_local[MAX_SIZE][MAX_DIM];
#pragma HLS ARRAY_PARTITION variable=in_local type=complete dim=2
```

Example 4

Partitioned arrays can be addressed in your code by the new structure of the array, as shown in the following code example;

```
struct SS
{
    int x[N];
    int y[N];
};

int top(SS *a, int b[4][6], SS &c) {
#pragma HLS array_partition type=complete dim=1 variable=b
#pragma HLS interface mode=ap_memory port = b[0]
#pragma HLS interface mode=ap_memory port = b[1]
#pragma HLS interface mode=ap_memory port = b[2]
#pragma HLS interface mode=ap_memory port = b[3]
```

See Also

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

pragma HLS array_reshape

Description



IMPORTANT! *Array_Partition and Array_Reshape pragmas and directives are not supported for M_AXI Interfaces on the top-level function. Instead you can use the `hls::vector` data types as described in [Vector Data Types](#).*

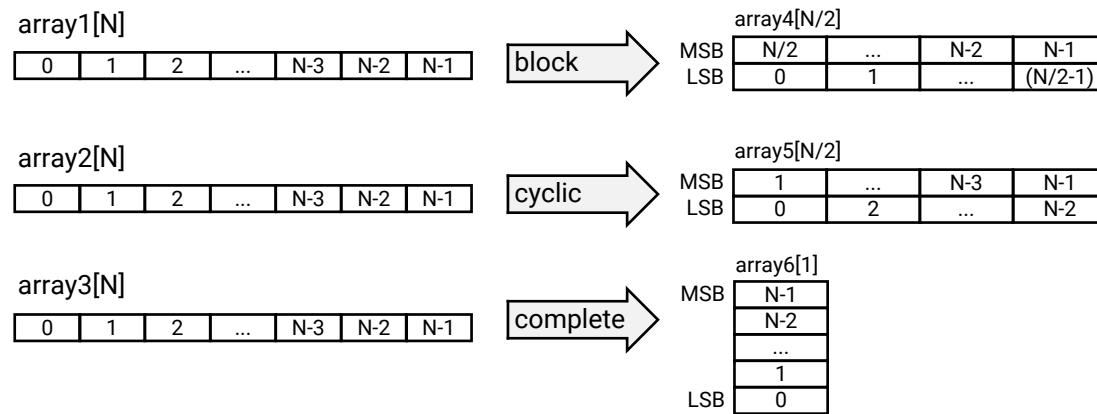
The `ARRAY_RESHAPE` pragma reforms the array with vertical remapping and concatenating elements of arrays by increasing bit-widths. This reduces the number of block RAM consumed while providing parallel access to the data. This pragma creates a new array with fewer elements but with greater bit-width, allowing more data to be accessed in a single clock cycle.

Given the following code:

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

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

Figure 123: ARRAY_RESHAPE Pragma



X14307-110217

Syntax

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

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

Where:

- `variable=<name>`: Required argument that specifies the array variable to be reshaped.
- `type=<type>`: Optionally specifies the partition type. The default type is `complete`. The following types are supported:

- **cyclic**: Cyclic reshaping creates smaller arrays by interleaving elements from the original array. For example, if `factor=3` is used, element 0 is assigned to the first new array, element 1 to the second new array, element 2 is assigned to the third new array, and then element 3 is assigned to the first new array again. The final array is a vertical concatenation (word concatenation, to create longer words) of the new arrays into a single array.
- **block**: Block reshaping creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into $<N>$ equal blocks where $<N>$ is the integer defined by `factor=`, and then combines the $<N>$ blocks into a single array with `word-width*N`.
- **complete**: Complete reshaping decomposes the array into temporary individual elements and then recombines them into an array with a wider word. For a one-dimension array this is equivalent to creating a very-wide register (if the original array was N elements of M bits, the result is a register with $N*M$ bits). This is the default type of array reshaping.
- **factor=<int>**: Specifies the amount to divide the current array by (or the number of temporary arrays to create). A factor of 2 splits the array in half, while doubling the bit-width. A factor of 3 divides the array into three, with triple the bit-width.



IMPORTANT! For complete type partitioning, the `factor` is not specified. For block and cyclic reshaping, the `factor=` is required.

- **dim=<int>**: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to $<N>$, for an array with $<N>$ dimensions:
 - If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
 - Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.
- **object**: A keyword relevant for container arrays only. When the keyword is specified the `ARRAY_reshape` pragma applies to the objects in the container, reshaping all dimensions of the objects within the container, but all dimensions of the container itself are preserved. When the keyword is not specified the pragma applies to the container array and not the objects.

Example 1

Reshapes an 8-bit array with 17 elements, `AB[17]`, into a new 32-bit array with five elements using block mapping.

```
#pragma HLS array_reshape variable=AB type=block factor=4
```



TIP: `factor=4` indicates that the array should be divided into four; this means that 17 elements are reshaped into an array of five elements, with four times the bit-width. In this case, the last element, `AB[17]`, is mapped to the lower eight bits of the fifth element, and the rest of the fifth element is empty.

Example 2

Reshapes the two-dimensional array `AB [6] [4]` into a new array of dimension `[6][2]`, in which dimension 2 has twice the bit-width.

```
#pragma HLS array_reshape variable=AB type=block factor=2 dim=2
```

Example 3

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

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



TIP: `dim=0` means to reshape all dimensions of the array.

Example 4

Partitioned arrays can be addressed in your code by the new structure of the array, as shown in the following code example;

```
struct SS
{
    int x[N];
    int y[N];
};

int top(SS *a, int b[4][6], SS &c) {
#pragma HLS array_reshape type=complete dim=0 variable=b
#pragma HLS interface mode=ap_memory port=b[0]
```

See Also

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

pragma HLS bind_op

Description

Vitis HLS implements the operations in the code using specific implementations. The `BIND_OP` pragma specifies that for a specific variable, an operation (`mul`, `add`, `div`) should be mapped to a specific device resource for implementation (`impl`) in the RTL. If the `BIND_OP` pragma is not specified, Vitis HLS automatically determines the resources to use for operations.

For example, to indicate that a specific multiplier operation (`mul`) is implemented in the device fabric rather than a DSP, you can use the `BIND_OP` pragma.

You can also specify the latency of the operation using the `latency` option.

IMPORTANT! To use the `latency` option, the operation must have an available multi-stage implementation. The HLS tool provides a multi-stage implementation for all basic arithmetic operations (add, subtract, multiply, and divide), and all floating-point operations.

Syntax

Place the pragma in the C source within the body of the function where the variable is defined.

```
#pragma HLS bind_op variable=<variable> op=<type>\impl=<value> latency=<int>
```

Where:

- `variable=<variable>`: Defines the variable to assign the `BIND_OP` pragma to. The variable in this case is one that is assigned the result of the operation that is the target of this pragma.
- `op=<type>`: Defines the operation to bind to a specific implementation resource. Supported functional operations include: `mul`, `add`, and `sub`. Supported floating point operations include: `fadd`, `fsub`, `fdiv`, `fexp`, `flog`, `fmul`, `frsqrt`, `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 33: Supported Combinations of Functional Operations, Implementation, and Latency

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

Table 33: Supported Combinations of Functional Operations, Implementation, and Latency (cont'd)

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

Table 34: Supported Combinations of Floating Point Operations, Implementation, and Latency

Operation	Implementation	Min Latency	Max Latency
fadd	fabric	0	13
fadd	fulldsp	0	12
fadd	primitivedsp	0	3
fsub	fabric	0	13
fsub	fulldsp	0	12
fsub	primitivedsp	0	3
fdiv	fabric	0	29
fexp	fabric	0	24
fexp	meddsp	0	21
fexp	fulldsp	0	30
flog	fabric	0	24
flog	meddsp	0	23
flog	fulldsp	0	29
fmul	fabric	0	9
fmul	meddsp	0	9
fmul	fulldsp	0	9
fmul	maxdsp	0	7
fmul	primitivedsp	0	4
fsqrt	fabric	0	29
frsqrt	fabric	0	38
frsqrt	fulldsp	0	33
frecip	fabric	0	37
frecip	fulldsp	0	30
dadd	fabric	0	13
dadd	fulldsp	0	15
dsub	fabric	0	13
dsub	fulldsp	0	15
ddiv	fabric	0	58
dexp	fabric	0	40
dexp	meddsp	0	45

Table 34: Supported Combinations of Floating Point Operations, Implementation, and Latency (cont'd)

Operation	Implementation	Min Latency	Max Latency
dexp	fulldsp	0	57
dlog	fabric	0	38
dlog	meddsp	0	49
dlog	fulldsp	0	65
dmul	fabric	0	10
dmul	meddsp	0	13
dmul	fulldsp	0	13
dmul	maxdsp	0	14
dsqrt	fabric	0	58
drsqrt	fulldsp	0	111
drecip	fulldsp	0	36
hadd	fabric	0	9
hadd	meddsp	0	12
hadd	fulldsp	0	12
hsub	fabric	0	9
hsub	meddsp	0	12
hsub	fulldsp	0	12
hdiv	fabric	0	16
hmul	fabric	0	7
hmul	fulldsp	0	7
hmul	maxdsp	0	9
hsqrt	fabric	0	16

Example

In the following example, a two-stage pipelined multiplier using fabric logic is specified to implement the multiplication for variable `c` of the function `foo`.

```
int foo (int a, int b) {
int c, d;
#pragma HLS BIND_OP variable=c op=mul impl=fabric latency=2
c = a*b;
d = a*c;
return d;
}
```



TIP: The HLS tool selects the implementation to use for variable `d`.

See Also

- [set_directive_bind_op](#)

- [pragma HLS bind_storage](#)
-

pragma HLS bind_storage

Description

The BIND_STORAGE pragma assigns a variable (array, or function argument) in the code to a specific memory type (`type`) in the RTL. If the pragma is not specified, the Vitis HLS tool determines the memory type to assign. The HLS tool implements the memory using specified implementations (`impl`) in the hardware.

For example, you can use the pragma to specify which memory type to use to implement an array. This lets you control whether the array is implemented as a single or a dual-port RAM for example. Also, this allows you to control whether the array is implemented as a single or a dual-port RAM.

 **IMPORTANT!** This feature is important for arrays on the top-level function interface, because the memory type associated with the array determines the number and type of ports needed in the RTL, as discussed in [Arrays on the Interface](#). However, for variables assigned to top-level function arguments you must assign the memory type and implementation using the `-storage_type` and `-storage_impl` options of the INTERFACE pragma or directive.

You can also specify the latency of the implementation. For block RAMs on the interface, the `latency` option lets you model off-chip, non-standard SRAMs at the interface, for example supporting an SRAM with a latency of 2 or 3. For internal operations, the `latency` option allows the memory to be implemented using more pipelined stages. These additional pipeline stages can help resolve timing issues during RTL synthesis.

 **IMPORTANT!** To use the `latency` option, the operation must have an available multi-stage implementation. The HLS tool provides a multi-stage implementation for all block RAMs.

Syntax

Place the pragma in the C/C++ source within the body of the function where the variable is defined.

```
#pragma HLS bind_storage variable=<variable> type=<type>\  
[ impl=<value> latency=<int> ]
```

Where:

- **variable=<variable>**: Defines the variable to assign the BIND_STORAGE pragma to. This is required when specifying the pragma.



TIP: If the variable is an argument of a top-level function, then use the `-storage_type` and `-storage_impl` options of the INTERFACE pragma or directive.

- **type=<type>:** Defines the type of memory to bind to the specified variable. Supported types include: `fifo`, `ram_1p`, `ram_1wnr`, `ram_2p`, `ram_s2p`, `ram_t2p`, `rom_1p`, `rom_2p`, `rom_np`.

Table 35: Storage Types

Type	Description
FIFO	A FIFO. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
RAM_1P	A single-port RAM. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
RAM_1WNR	A RAM with 1 write port and N read ports, using N banks internally.
RAM_2P	A dual-port RAM that allows read operations on one port and both read and write operations on the other port.
RAM_S2P	A dual-port RAM that allows read operations on one port and write operations on the other port.
RAM_T2P	A true dual-port RAM with support for both read and write on both ports.
ROM_1P	A single-port ROM. Vitis HLS determines how to implement this in the RTL, unless the <code>-impl</code> option is specified.
ROM_2P	A dual-port ROM.
ROM_NP	A multi-port ROM.

- **impl=<value>:** Defines the implementation for the specified memory type. Supported implementations include: `bram`, `bram_ecc`, `lutram`, `uram`, `uram_ecc`, `srl`, `memory`, and `auto` as described below.

Table 36: Supported Implementation

Name	Description
MEMORY	Generic memory lets the Vivado tool choose the implementation.
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.

Table 37: Supported Implementations by FIFO/RAM/ROM

Type	Command/Pragma	Scope	Supported Implementations
FIFO	bind_storage ¹	local	AUTO , BRAM, LUTRAM, URAM, MEMORY, SRL
FIFO	config_storage	global	AUTO , BRAM, LUTRAM, URAM, MEMORY, SRL
RAM* ROM*	bind_storage	local	AUTO BRAM, BRAM_ECC, LUTRAM, URAM, URAM_ECC
RAM* ROM*	config_storage ²	global	N/A
RAM_1P	set_directive_interface s_axilite - storage_impl	local	AUTO , BRAM, URAM
	config_interface - m_axi_buffer_impl	global	AUTO , BRAM , LUTRAM, URAM

- When no implementation is specified the directive uses AUTOSRL behavior as a default. However, this value cannot be specified.
 - config_storage only supports FIFO types.
- latency=<int>**: Defines the default latency for the binding of the type. As shown in the following table, the valid latency varies according to the specified type and impl. The default is -1, which lets Vitis HLS choose the latency.

Table 38: Supported Combinations of Memory Type, Implementation, and Latency

Type	Implementation	Min Latency	Max Latency
FIFO	BRAM	0	0
FIFO	LUTRAM	0	0
FIFO	MEMORY	0	0
FIFO	SRL	0	0
FIFO	URAM	0	0
RAM_1P	AUTO	1	3
RAM_1P	BRAM	1	3
RAM_1P	LUTRAM	1	3
RAM_1P	URAM	1	3
RAM_1WNR	AUTO	1	3
RAM_1WNR	BRAM	1	3
RAM_1WNR	LUTRAM	1	3
RAM_1WNR	URAM	1	3
RAM_2P	AUTO	1	3
RAM_2P	BRAM	1	3
RAM_2P	LUTRAM	1	3
RAM_2P	URAM	1	3

Table 38: Supported Combinations of Memory Type, Implementation, and Latency (cont'd)

Type	Implementation	Min Latency	Max Latency
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
ROM_NP	LUTRAM	1	3

IMPORTANT! Any combinations of memory type and implementation that are not listed in the prior table are not supported by `set_directive_bind_storage`.

Example

The pragma specifies that the variable `coeffs` uses a single port RAM implemented on a BRAM core from the library.

```
#pragma HLS bind_storage variable=coeffs type=RAM_1P impl=bram
```



TIP: The ports created in the RTL to access the values of `coeffs` are defined in the RAM_1P.

See Also

- [set_directive_bind_storage](#)
- [pragma HLS bind_op](#)

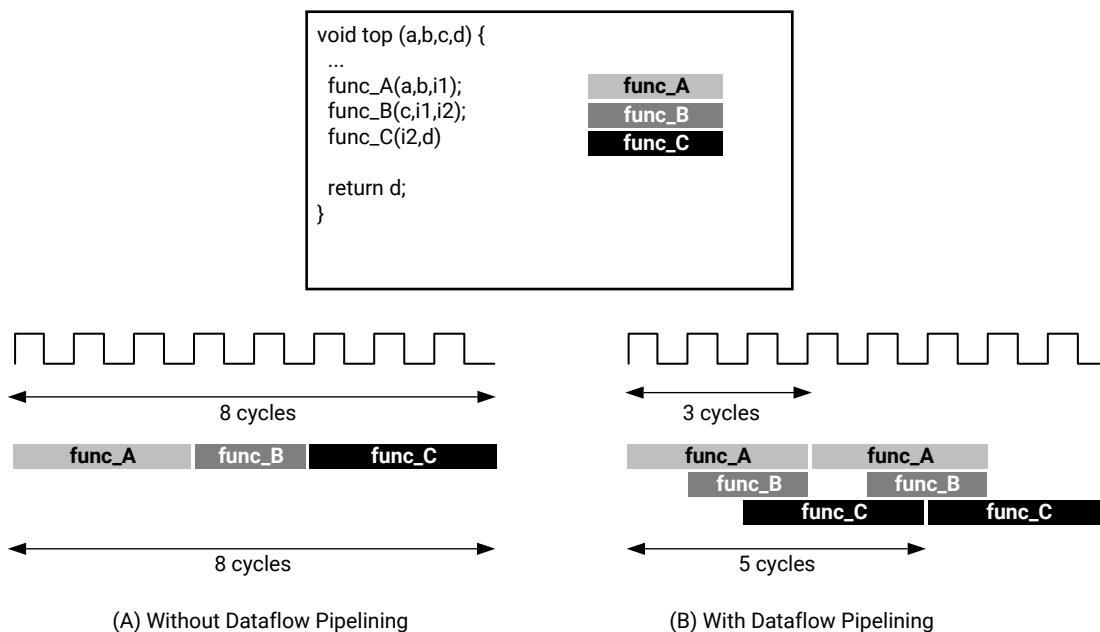
pragma HLS dataflow

Description

The DATAFLOW pragma enables task-level pipelining 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 124: DATAFLOW Pragma



X14266-110217

When the DATAFLOW pragma is specified, the HLS tool analyzes the dataflow between sequential functions or loops and creates channels (based on ping pong RAMs or FIFOs) that allow consumer functions or loops to start operation before the producer functions or loops have completed. This allows functions or loops to operate in parallel, which decreases latency and improves the throughput of the RTL.

If no initiation interval (number of cycles between the start of one function or loop and the next) is specified, the HLS tool attempts to minimize the initiation interval and start operation as soon as data is available.



TIP: The `config_dataflow` command specifies the default memory channel and FIFO depth used in dataflow optimization.

For the DATAFLOW optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent the HLS tool from performing the DATAFLOW optimization. Refer to [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=[array | pointer]**: Optionally specifies a class of variables in which the dependence needs clarification. Valid values include `array` or `pointer`.



TIP: `<class>` and `variable=` should not be specified together as you can specify dependence for a variable, or a class of variables within a function.

- **type=[inter | intra]**: Valid values include `intra` or `inter`. Specifies whether the dependence is:

- **intra**: Dependence within the same loop iteration. When dependence `<type>` is specified as `intra`, and `<dependent>` is false, the HLS tool might move operations freely within a loop, increasing their mobility and potentially improving performance or area. When `<dependent>` is specified as true, the operations must be performed in the order specified.

- **inter**: dependence between different loop iterations. This is the default `<type>`. If dependence `<type>` is specified as `inter`, and `<dependent>` is false, it allows the HLS tool to perform operations in parallel if the function or loop is pipelined, or the loop is unrolled, or partially unrolled, and prevents such concurrent operation when `<dependent>` is specified as true.

- **direction=[RAW | WAR | WAW]**: This is relevant for loop-carry dependencies only, and specifies the direction for a dependence:

- **RAW (Read-After-Write - true dependence)**: The write instruction uses a value used by the read instruction.
- **WAR (Write-After-Read - anti dependence)**: The read instruction gets a value that is overwritten by the write instruction.

- **WAW (Write-After-Write - output dependence):** Two write instructions write to the same location, in a certain order.
- **distance=<int>:** Specifies the inter-iteration distance for array access. Relevant only for loop-carry dependencies where dependence is set to `true`.
- **dependent=[true | false]:** This argument should be specified to indicate whether a dependence is `true` and needs to be enforced, or is `false` and should be removed. However, when not specified, the tool will return a warning that the value was not specified and will assume a value of `false`.

Example 1

In the following example, the HLS tool does not have any knowledge about the value of `cols` and conservatively assumes that there is always a dependence between the write to `buff_A[1][col]` and the read from `buff_A[1][col]`. In an algorithm such as this, it is unlikely `cols` will ever be zero, but the HLS tool cannot make assumptions about data dependencies. To overcome this deficiency, you can use the `DEPENDENCE` pragma to state that there is no dependence between loop iterations (in this case, for both `buff_A` and `buff_B`).

```
void foo(int rows, int cols, ...)
    for (row = 0; row < rows + 1; row++) {
        for (col = 0; col < cols + 1; col++) {
            #pragma HLS PIPELINE II=1
            #pragma HLS dependence variable=buf_A type=inter dependent=false
            #pragma HLS dependence variable=buf_B type=inter dependent=false
            if (col < cols) {
                buf_A[2][col] = buf_A[1][col]; // read from buf_A[1][col]
                buf_A[1][col] = buf_A[0][col]; // write to buf_A[1][col]
                buf_B[1][col] = buf_B[0][col];
                temp = buf_A[0][col];
            }
        }
    }
}
```

Example 2

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

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

Example 3

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

```
#pragma HLS dependence class=array type=intra direction=RAW dependent=true
```

See Also

- [set_directive_dependence](#)
- [pragma HLS disaggregate](#)

- [pragma HLS pipeline](#)
-

pragma HLS disaggregate

Description

The DISAGGREGATE pragma lets you deconstruct a `struct` variable into its individual elements. The number and type of elements created are determined by the contents of the struct itself.



IMPORTANT! Structs used as arguments to the top-level function are aggregated by default, but can be disaggregated with this directive or pragma. Refer to [AXI4-Stream Interfaces](#) for important information about disaggregating structs associated with streams.

Syntax

Place the pragma in the C source within the boundaries of the region, function, or loop.

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

Options

Where:

- `variable=<variable>`: Specifies the struct variable to disaggregate.

Example 1

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

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

Example 2

Disaggregated structs can be addressed in your code by the using standard C/C++ coding style as shown below. Notice the different methods for accessing the pointer element (`a`) versus the reference element (`c`);

```
struct SS
{
    int x[N];
    int y[N];
};

int top(SS *a, int b[4][6], SS &c) {
#pragma HLS disaggregate variable = a
#pragma HLS interface s_axilite port = a->x
#pragma HLS interface s_axilite port = a->y
```

```
// Following is now supported
#pragma HLS disaggregate variable = c
#pragma HLS interface ap_memory port = c.x
#pragma HLS interface ap_memory port = c.y
```

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

```
func_sub (p, q) {
#pragma HLS inline off
int q1 = q + 10;
func(p1,q); // foo_3
...
}
void func_top { a, b, c, d} {
#pragma HLS inline recursive
...
func(a,b); // func_1
func(a,c); // func_2
func_sub(a,d);
...
}
```



TIP: *Notice in this example that `INLINE RECURSIVE` applies downward to the contents of function `func_top`, but `INLINE OFF` applies to `func_sub` directly.*

Example 2

This example inlines the `copy_output` function into any functions or regions calling `copy_output`.

```
void copy_output(int *out, int out_lcl[OSize * OSize], int output) {
    #pragma HLS INLINE
    // Calculate each work_item's result update location
    int stride = output * OSize * OSize;

    // Work-item updates output filter/image in DDR
    writeOut: for(int itr = 0; itr < OSize * OSize; itr++) {
        #pragma HLS PIPELINE
        out[stride + itr] = out_lcl[itr];
    }
}
```

See Also

- [set_directive_inline](#)
- [pragma HLS allocation](#)

pragma HLS interface

Description

In C/C++ code, all input and output operations are performed, in zero time, through formal function arguments. In a RTL design, these same input and output operations must be performed through a port in the design interface and typically operate using a specific input/output (I/O) protocol. For more information, see [Defining Interfaces](#).

The INTERFACE pragma specifies how RTL ports are created from the function definition during interface synthesis. The ports in the RTL implementation are derived from the following:

- Block-level I/O protocols: Provide signals to control when the function starts operation, and indicate when function operation ends, is idle, and is ready for new inputs. The implementation of a block-level protocol is:
 - Specified by the `<mode>` values `ap_ctrl_none`, `ap_ctrl_hs`, or `ap_ctrl_chain`. The `ap_ctrl_chain` block protocol is the default.
 - Associated with the function name.
- Function arguments: Each function argument can be specified to have its own port-level (I/O) interface protocol, such as valid handshake (`ap_vld`), or acknowledge handshake (`ap_ack`). Port-level interface protocols are created for each argument in the top-level function and the function return, if the function returns a value. The default I/O protocol created depends on the type of C argument. After the block-level protocol has been used to start the operation of the block, the port-level I/O protocols are used to sequence data into and out of the block.

- 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. Refer to [AXI Burst Transfers](#) for more information.

For example, pipelined accesses from a `for`-loop of 100 iterations will not fill the max burst length when `max_read_burst_length` or `max_write_burst_length` is set to 128.

However, if the design is doing longer accesses than the specified maximum burst length, the access will be split into multiple bursts. For example, a pipelined `for`-loop with 100 accesses, and `max_read_burst_length` or `max_write_burst_length` of 64, will be split into 2 transactions: one sized to the max burst length (64), and one with the remaining data (burst of length 36 words).

Syntax

Place the pragma within the boundaries of the function.

```
#pragma HLS interface mode=<mode> port=<name> bundle=<string> \
register register_mode=<mode> depth=<int> offset=<string> latency=<value> \
clock=<string> name=<string> storage_type=<value> \
num_read_outstanding=<int> num_write_outstanding=<int> \
max_read_burst_length=<int> max_write_burst_length=<int>
```

Where:

- `mode=<mode>`: Specifies the interface protocol mode for function arguments, global variables used by the function, or the block-level control protocols. The mode can be specified as one of the following:
 - `ap_none`: No protocol. The interface is a data port.
 - `ap_stable`: No protocol. The interface is a data port. The HLS tool assumes the data port is always stable after reset, which allows internal optimizations to remove unnecessary registers.
 - `ap_vld`: Implements the data port with an associated `valid` port to indicate when the data is valid for reading or writing.
 - `ap_ack`: Implements the data port with an associated `acknowledge` port to acknowledge that the data was read or written.
 - `ap_hs`: Implements the data port with associated `valid` and `acknowledge` ports to provide a two-way handshake to indicate when the data is valid for reading and writing and to acknowledge that the data was read or written.
 - `ap_ovld`: Implements the output data port with an associated `valid` port to indicate when the data is valid for reading or writing.



IMPORTANT! The HLS tool implements the input argument or the input half of any read/write arguments with mode `ap_none`.

- `ap_fifo`: Implements the port with a standard FIFO interface using data input and output ports with associated active-Low FIFO `empty` and `full` ports.

Note: You can only use this interface on read arguments or write arguments. The `ap_fifo` mode does not support bidirectional read/write arguments.

- **ap_memory:** Implements array arguments as a standard RAM interface. If you use the RTL design in the Vivado IP integrator, the memory interface appears as discrete ports.
- **bram:** Implements array arguments as a standard RAM interface. If you use the RTL design in the IP integrator, the memory interface appears as a single port.
- **axis:** Implements all ports as an AXI4-Stream interface.
- **s_axilite:** Implements all ports as an AXI4-Lite interface. The HLS tool produces an associated set of C driver files during the Export RTL process.
- **m_axi:** Implements all ports as an AXI4 interface. You can use the `config_interface` command to specify either 32-bit (default) or 64-bit address ports and to control any address offset.
- **ap_ctrl_chain:** Implements a set of block-level control ports to start the design operation, continue operation, and indicate when the design is idle, done, and ready for new input data.

Note: The `ap_ctrl_chain` interface mode is similar to `ap_ctrl_hs` but provides an additional input signal `ap_continue` to apply back pressure. Xilinx recommends using the `ap_ctrl_chain` block-level I/O protocol when chaining the HLS tool blocks together.

Note: The `ap_ctrl_chain` is the default block-level I/O protocol.

- **ap_ctrl_hs:** Implements a set of block-level control ports to start the design operation and to indicate when the design is idle, done, and ready for new input data.
- **ap_ctrl_none:** No block-level I/O protocol.

Note: Using the `ap_ctrl_none` mode might prevent the design from being verified using the C//RTL co-simulation feature.

- **port=<name>:** Specifies the name of the function argument, 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 memory mapped (`m_axi`) interfaces for the specified port.
 - In an `s_axilite` interface, `<string>` specifies the address in the register map.
 - In an `m_axi` interface this option overrides the global option specified by the `config_interface -m_axi_offset` option, and `<string>` is specified as:
 - `off`: Do not generate an offset port.
 - `direct`: Generate a scalar input offset port.
 - `slave`: Generate an offset port and automatically map it to an AXI4-Lite slave interface. This is the default offset.

- `clock=<name>`: Optionally specified only for interface mode `s_axilite`. This defines the clock signal to use for the interface. By default, the AXI4-Lite interface clock is the same clock as the system clock. This option is used to specify a separate clock for the AXI4-Lite (`s_axilite`) interface.



TIP: If the `bundle` option is used to group multiple top-level function arguments into a single AXI4-Lite interface, the `clock` option need only be specified on one of the `bundle` members.

- `name=<string>`: Specifies a name for the port which will be used in the generated RTL.
- `latency=<value>`: When `mode` is `m_axi`, this specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (latency) before the read or write is expected. If this figure is too low, the design will be ready too soon and might stall waiting for the bus. If this figure is too high, bus access can be granted but the bus might stall waiting on the design to start the access.
- `storage_impl=<impl>`: For use with `s_axilite` only. This options defines a storage implementation to assign to the interface. Supported implementation values include `auto`, `bram`, and `uram`. The default is `auto`.



TIP: `uram` is a synchronous memory with only a single clock for two ports. Therefore `uram` cannot be specified for an `s_axilite` adapter with a second clock.

- `storage_type=<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`.

- `-max_widen_bitwidth <int>`: Specifies the maximum bit width available for the interface when automatically widening the interface. This overrides the global value specified by the `config_interface -m_axi_max_bitwidth` command.

Example 1

In this example, both function arguments are implemented using an AXI4-Stream interface:

```
void example(int A[50], int B[50]) {
    //Set the HLS native interface types
    #pragma HLS INTERFACE mode=axis port=A
    #pragma HLS INTERFACE mode=axis port=B
    int i;
    for(i = 0; i < 50; i++){
        B[i] = A[i] + 5;
    }
}
```

Example 2

The following turns off block-level I/O protocols, and is assigned to the function return value:

```
#pragma HLS interface mode=ap_ctrl_none port=return
```

The function argument `InData` is specified to use the `ap_vld` interface and also indicates the input should be registered:

```
#pragma HLS interface mode=ap_vld register port=InData
```

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

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

```
#pragma HLS INTERFACE mode=s_axilite port=input bundle=control
#pragma HLS INTERFACE mode=s_axilite port=output bundle=control
#pragma HLS INTERFACE mode=s_axilite port=return bundle=control

#pragma HLS dataflow
```

See Also

- [set_directive_interface](#)
- [pragma HLS bind_storage](#)

pragma HLS latency

Description

Specifies a minimum or maximum latency value, or both, for the completion of functions, loops, and regions.

- **Latency:** Number of clock cycles required to produce an output.
- **Function latency:** Number of clock cycles required for the function to compute all output values, and return.
- **Loop latency:** Number of cycles to execute all iterations of the loop.

Vitis HLS always tries to minimize latency in the design. When the LATENCY pragma is specified, the tool behavior is as follows:

- Latency is greater than the minimum, or less than the maximum: The constraint is satisfied. No further optimizations are performed.
- Latency is less than the minimum: If the HLS tool can achieve less than the minimum specified latency, it extends the latency to the specified value, potentially enabling increased sharing.
- Latency is greater than the maximum: If the HLS tool cannot schedule within the maximum limit, it increases effort to achieve the specified constraint. If it still fails to meet the maximum latency, it issues a warning, and produces a design with the smallest achievable latency in excess of the maximum.



TIP: You can also use the LATENCY pragma to limit the efforts of the tool to find an optimum solution.

Specifying latency constraints for scopes within the code: loops, functions, or regions, reduces the possible solutions within that scope and can improve tool runtime. Refer to [Improving Synthesis Runtime and Capacity](#) for more information.

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 $<N>$ times.
- However, part of the loop body is enabled by a conditional statement, and as a result only executes $<M>$ times, where $<N>$ is an integer multiple of $<M>$.
- The conditional code has an occurrence that is N/M times slower than the rest of the loop body.

For example, in a loop that executes 10 times, a conditional statement within the loop only executes two times has an occurrence of 5 (or 10/2).

Identifying a region with the OCCURRENCE pragma allows the functions and loops in that region to be pipelined with a higher initiation interval that is slower than the enclosing function or loop.

Syntax

Place the pragma in the C source within a 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! $<N>$ must be an integer multiple of $<M>$.

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. An II of 1 processes a new input every clock cycle. You can specify the initiation interval through the use of the II option for the pragma.

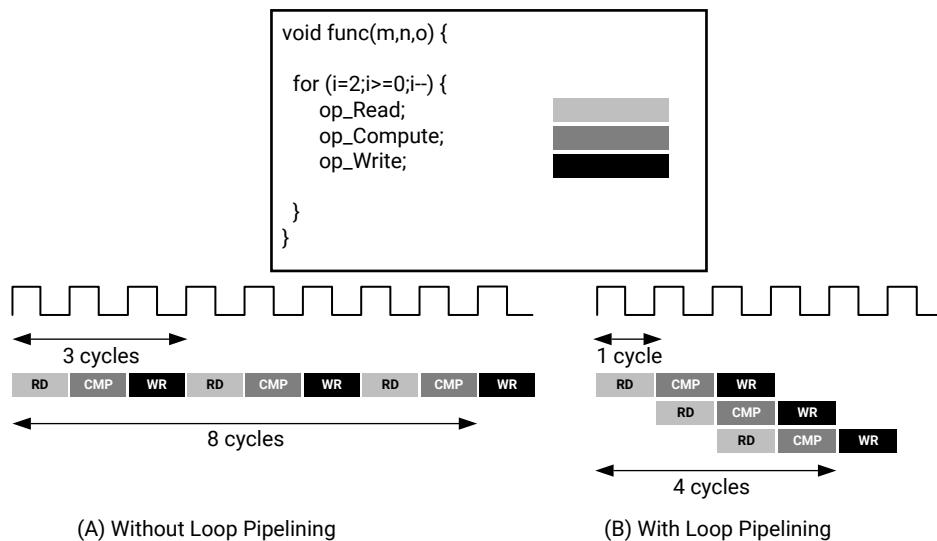
As a default behavior, with the PIPELINE pragma or directive Vitis HLS will generate the minimum II for the design according to the specified clock period constraint. The emphasis will be on meeting timing, rather than on achieving II unless the II option is specified.

If the Vitis HLS tool cannot create a design with the specified II, it issues a warning and creates a design with the lowest possible II.

You can then analyze this design with the warning message to determine what steps must be taken to create a design that satisfies the required initiation interval.

Pipelining a loop allows the operations of the loop to be implemented in a concurrent manner as shown in the following figure. In the figure, (A) shows the default sequential operation where there are three clock cycles between each input read (II=3), and it requires eight clock cycles before the last output write is performed. (B) shows the pipelined operations that show one cycle between reads (II=1), and 4 cycles to the last write.

Figure 125: Loop Pipeline



X14277-110217

IMPORTANT! Loop carry dependencies can prevent pipelining. Use the DEPENDENCE pragma or directive to provide additional information to overcome loop-carry dependencies, and allow loops to be pipelined (or pipelined with lower intervals).

Syntax

Place the pragma in the C source within the body of the function or loop.

```
#pragma HLS pipeline II=<int> off rewind style=<value>
```

Where:

- **II=<int>**: Specifies the desired initiation interval for the pipeline. The HLS tool tries to meet this request. Based on data dependencies, the actual result might have a larger initiation interval.
- **off**: Optional keyword. Turns off pipeline for a specific loop or function. This can be used to disable pipelining for a specific loop when `config_compile -pipeline_loops` is used to globally pipeline loops.
- **rewind**: Optional keyword. Enables rewinding as described in [Rewinding Pipelined Loops for Performance](#). This enables continuous loop pipelining with no pause between one execution of the loop ending and the next execution starting. Rewinding is effective only if there is one single loop (or a perfect loop nest) inside the top-level function. The code segment before the loop:
 - Is considered as initialization.
 - Is executed only once in the pipeline.
 - Cannot contain any conditional operations (if-else).



TIP: This feature is only supported for pipelined loops; it is not supported for pipelined functions.

- **style=<stp | frp | 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 `func` is pipelined with an initiation interval of 1.

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

See Also

- [set_directive_pipeline](#)
- [pragma HLS dependence](#)
- [config_compile](#)

pragma HLS protocol

Description

This command specifies a region of code, a protocol region, in which no clock operations will be inserted by Vitis HLS unless explicitly specified in the code. Vitis HLS will not insert any clocks between operations in the region, including those which read from or write to function arguments. The order of read and writes will therefore be strictly followed in the synthesized RTL.

A region of code can be created in the C/C++ code by enclosing the region in braces "{}" and naming it. The following defines a region named `io_section`:

```
io_section:{  
...  
lines of code  
...  
}
```

A clock operation can be explicitly specified in C/C++ code using an `ap_wait()` statement, and may be specified in C++ code by using the `wait()` statement. The `ap_wait` and `wait` statements have no effect on the simulation of the design.

Syntax

Place the pragma in the C source within the body of the function or protocol region.

```
#pragma HLS protocol [floating | fixed]
```

Options

- **floating:** Lets code statements outside the protocol region overlap and execute in parallel with statements in the protocol region in the final RTL. The protocol region remains cycle accurate, but outside operations can occur at the same time. This is the default mode.
- **fixed:** The fixed mode ensures that statements outside the protocol region do not execute in parallel with the protocol region.

Examples

This example defines a protocol region, `io_section` in function `foo` where the pragma defines that region as a floating protocol region as the default mode:

```
io_section: {  
#pragma HLS protocol  
...  
}
```

See Also

- [set_directive_protocol](#)
-

pragma HLS reset

Description

Adds or removes resets for specific state variables (global or static).

The reset port is used to restore the registers and block RAM, connected to the port, to an initial value any time the reset signal is applied. The presence and behavior of the RTL reset port is controlled using the `config_rtl` settings. The reset settings include the ability to set the polarity of the reset, and specify whether the reset is synchronous or asynchronous, but more importantly it controls, through the `reset` option, which registers are reset when the reset signal is applied. For more information, see [Controlling 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 stable

Description

The STABLE pragma is applied to arguments of a DATAFLOW or PIPELINE region and is used to indicate that an input or output of this region can be ignored when generating the synchronizations at entry and exit of the DATAFLOW region. This means that the reading processes (resp. read accesses) of that argument do not need to be part of the “first stage” of the task-level (resp. fine-grain) pipeline for inputs, and the writing process (resp. write accesses) do not need to be part of the last stage of the task-level (resp. fine-grain) pipeline for outputs.

The pragma can be specified at any point in the hierarchy, on a scalar or an array, and automatically applies to all the DATAFLOW or PIPELINE regions below that point. The effect of STABLE for an input is that a DATAFLOW or PIPELINE region can start another iteration even though the value of the previous iteration has not been read yet. For an output, this implies that a write of the next iteration can occur although the previous iteration is not done.

Syntax

```
#pragma HLS stable variable=<a>
```

Where:

- **variable=<a>**: Specifies the variable to which the STABLE pragma is applied.

Examples

In the following example, without the STABLE pragma, `proc1` and `proc2` would be synchronized to acknowledge the reading of their inputs (including `A`). With the pragma, `A` is no longer considered as an input that needs synchronization.

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

See Also

- [set_directive_stable](#)
- [pragma HLS dataflow](#)
- [pragma HLS pipeline](#)

pragma HLS stream

Description

By default, array variables are implemented as RAM:

- Top-level function array parameters are implemented as a RAM interface port.
- General arrays are implemented as RAMs for read-write access.
- Arrays involved in sub-functions, or loop-based DATAFLOW optimizations are implemented as a RAM ping pong buffer channel.

If the data stored in the array is consumed or produced in a sequential manner, a more efficient communication mechanism is to use streaming data as specified by the STREAM pragma, where FIFOs are used instead of RAMs.



IMPORTANT! When an argument of the top-level function is specified as INTERFACE type `ap_fifo`, the array is automatically implemented as streaming. See [Defining Interfaces](#) for more information.

Syntax

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

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

Where:

- **variable=<variable>**: Specifies the name of the array to implement as a streaming interface.
- **depth=<int>**: Relevant only for array streaming in DATAFLOW channels. By default, the depth of the FIFO implemented in the RTL is the same size as the array specified in the C code. This option lets you modify the size of the FIFO to specify a different depth.

When the array is implemented in a DATAFLOW region, it is common to use the `depth` option to reduce the size of the FIFO. For example, in a DATAFLOW region when all loops and functions are processing data at a rate of `II=1`, there is no need for a large FIFO because data is produced and consumed in each clock cycle. In this case, the `depth` option can be used to reduce the FIFO size to 1 to substantially reduce the area of the RTL design.



TIP: The `config_dataflow -depth` command provides the ability to stream all arrays in a DATAFLOW region. The `depth` option specified in the STREAM pragma overrides the `config_dataflow -depth` setting for the specified `<variable>`.

- **type=<arg>**: Specify a mechanism to select between FIFO, PIPO, synchronized shared (`shared`), and un-synchronized shared (`unsync`). The supported types include:
 - `fifo`: A FIFO buffer with the specified `depth`.
 - `piro`: A regular Ping-Pong buffer, with as many “banks” as the specified depth (default is 2).
 - `shared`: A shared channel, synchronized like a regular Ping-Pong buffer, with depth, but without duplicating the array data. Consistency can be ensured by setting the depth small enough, which acts as the distance of synchronization between the producer and consumer.



TIP: The default depth for `shared` is 1.

- `unsync`: Does not have any synchronization except for individual memory reads and writes. Consistency (read-write and write-write order) must be ensured by the design itself.

Example 1

The following example specifies array `A[10]` to be streaming, and implemented as a FIFO.

```
#pragma HLS STREAM variable=A
```

Example 2

In this example, array `B` is set to streaming with a FIFO depth of 12.

```
#pragma HLS STREAM variable=B depth=12 type=fifo
```

Example 3

Array C has streaming implemented as a PIPO.

```
#pragma HLS STREAM variable=C type=piro
```

See Also

- [set_directive_stream](#)
- [pragma HLS dataflow](#)
- [pragma HLS interface](#)
- [config_dataflow](#)

pragma HLS top

Description

Attaches a name to a function, which can then be used with the `set_top` command to synthesize the function and any functions called from the specified top-level. This is typically used to synthesize member functions of a class in C/C++.

Specify the TOP pragma in an active solution, and then use the `set_top` command with the new name.

Syntax

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

```
#pragma HLS top name=<string>
```

Where:

- `name=<string>`: Specifies the name to be used by the `set_top` command.

Examples

Function `foo_long_name` is designated the top-level function, and renamed to `DESIGN_TOP`. After the pragma is placed in the code, the `set_top` command must still be issued from the Tcl command line, or from the top-level specified in the IDE project settings.

```
void foo_long_name () {  
    #pragma HLS top name=DESIGN_TOP  
    ...  
}
```

Followed by the `set_top DESIGN_TOP` command.

See Also

- [set_directive_top](#)
 - [set_top](#)
-

pragma HLS unroll

Description

You can unroll loops to create multiple independent operations rather than a single collection of operations. The UNROLL pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel.

Loops in the C/C++ functions are kept rolled by default. When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. A loop is executed for the number of iterations specified by the loop induction variable. The number of iterations might also be impacted by logic inside the loop body (for example, `break` conditions or modifications to a loop exit variable). Using the UNROLL pragma you can unroll loops to increase data access and throughput.

The UNROLL pragma allows the loop to be fully or partially unrolled. Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently. Partially unrolling a loop lets you specify a factor N , to create N copies of the loop body and reduce the loop iterations accordingly.



TIP: To unroll a loop completely, the loop bounds must be known at compile time. This is not required for partial unrolling.

Partial loop unrolling does not require N to be an integer factor of the maximum loop iteration count. The Vitis HLS tool adds an exit check to ensure that partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```
for(int i = 0; i < X; i++) {  
    pragma HLS unroll factor=2  
    a[i] = b[i] + c[i];  
}
```

Loop unrolling by a factor of 2 effectively transforms the code to look like the following code where the `break` construct is used to ensure the functionality remains the same, and the loop exits at the appropriate point.

```
for(int i = 0; i < X; i += 2) {
    a[i] = b[i] + c[i];
    if (i+1 >= X) break;
    a[i+1] = b[i+1] + c[i+1];
}
```

In the example above, because the maximum iteration count, `X`, is a variable, the HLS tool might not be able to determine its value, so it adds an exit check and control logic to partially unrolled loops. However, if you know that the specified unrolling factor, 2 in this example, is an integer factor of the maximum iteration count `X`, the `skip_exit_check` option lets you remove the exit check and associated logic. This helps minimize the area and simplify the control logic.



TIP: When the use of pragmas like `ARRAY_PARTITION` or `ARRAY_RESHAPE` let more data be accessed in a single clock cycle, the HLS tool automatically unrolls any loops consuming this data, if doing so improves the throughput. The loop can be fully or partially unrolled to create enough hardware to consume the additional data in a single clock cycle. This automatic unrolling is controlled using the `config_unroll` command.

Syntax

Place the pragma in the C source within the body of the loop to unroll.

```
#pragma HLS unroll factor=<N> region skip_exit_check
```

Where:

- **factor=<N>**: Specifies a non-zero integer indicating that partial unrolling is requested. The loop body is repeated the specified number of times, and the iteration information is adjusted accordingly. If `factor=` is not specified, the loop is fully unrolled.
- **skip_exit_check**: Optional keyword that applies only if partial unrolling is specified with `factor=`. The elimination of the exit check is dependent on whether the loop iteration count is known or unknown:
 - **Fixed bounds**

No exit condition check is performed if the iteration count is a multiple of the factor.

If the iteration count is *not* an integer multiple of the factor, the tool:

 - Prevents unrolling.
 - Issues a warning that the exit check must be performed to proceed.
 - **Variable bounds**

The exit condition check is removed. You must ensure that:

- The variable bounds is an integer multiple of the factor.
- No exit check is in fact required.

Example 1

The following example fully unrolls `loop_1` in function `foo`. Place the pragma in the body of `loop_1` as shown.

```
loop_1: for(int i = 0; i < N; i++) {
    #pragma HLS unroll
    a[i] = b[i] + c[i];
}
```

Example 2

This example specifies an unroll factor of 4 to partially unroll `loop_2` of function `foo`, and removes the exit check.

```
void foo (...) {
    int8 array1[M];
    int12 array2[N];
    ...
loop_2: for(i=0;i<M;i++) {
    #pragma HLS unroll skip_exit_check factor=4
    array1[i] = ...;
    array2[i] = ...;
    ...
}
...
}
```

Example 3

The following example fully unrolls all loops inside `loop_1` in function `foo`, but not `loop_1` itself because the presence of the `region` keyword.

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {
    int temp1[N];
    loop_1: for(int i = 0; i < N; i++) {
        #pragma HLS unroll region
        temp1[i] = data_in[i] * scale;
        loop_2: for(int j = 0; j < N; j++) {
            data_out1[j] = temp1[j] * 123;
        }
        loop_3: for(int k = 0; k < N; k++) {
            data_out2[k] = temp1[k] * 456;
        }
    }
}
```

See Also

- [set_directive_unroll](#)

- [pragma HLS loop_flatten](#)
- [pragma HLS loop_merge](#)

Arbitrary Precision Data Types Library

C-based native data types are on 8-bit boundaries (8, 16, 32, 64 bits). RTL buses (corresponding to hardware) support arbitrary lengths. HLS needs a mechanism to allow the specification of arbitrary precision bit-width and not rely on the artificial boundaries of native C data types: if a 17-bit multiplier is required, you should not be forced to implement this with a 32-bit multiplier.

Vitis™ HLS provides both integer and fixed-point arbitrary precision data types for C++. The advantage of arbitrary precision data types is that they allow the C code to be updated to use variables with smaller bit-widths and then for the C simulation to be re-executed to validate that the functionality remains identical or acceptable.

Using Arbitrary Precision Data Types

Vitis HLS provides arbitrary precision integer data types that manage the value of the integer numbers within the boundaries of the specified width, as shown in the following table.

Table 39: Arbitrary Precision Data Types

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

The header files define the arbitrary precision types are also provided with Vitis HLS as a standalone package with the rights to use them in your own source code. The package, `xilinx_hls_lib_<release_number>.tgz`, is provided in the `include` directory in the Vitis HLS installation area.

Arbitrary Integer Precision Types with C++

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

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

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

```
#include "ap_int.h"

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



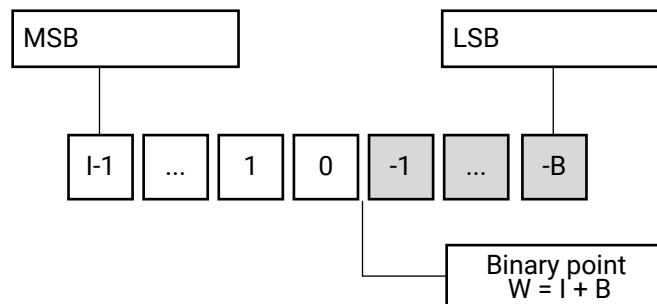
IMPORTANT! One disadvantage of AP data types is that arrays are not automatically initialized with a value of 0. You must manually initialize the array if desired.

Arbitrary Precision Fixed-Point Data Types

In Vitis HLS, it is important to use fixed-point data types, because the behavior of the C++ simulations performed using fixed-point data types match that of the resulting hardware created by synthesis. This allows you to analyze the effects of bit-accuracy, quantization, and overflow with fast C-level simulation.

These data types manage the value of real (non-integer) numbers within the boundaries of a specified total width and integer width, as shown in the following figure.

Figure 126: Fixed-Point Data Type



X14268-100620

Fixed-Point Identifier Summary

The following table provides a brief overview of operations supported by fixed-point types.

Table 40: Fixed-Point Identifier Summary

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

Notes:

- Using the AP_SAT* modes can result in higher resource usage as extra logic will be needed to perform saturation and this extra cost can be as high as 20% additional LUT usage.

Example Using ap_fixed

In this example the Vitis HLS `ap_fixed` type is used to define an 18-bit variable with 6 bits representing the numbers above the decimal point and 12-bits representing the value below the decimal point. The variable is specified as signed, the quantization mode is set to round to plus infinity and the default wrap-around mode is used for overflow.

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

C++ Arbitrary Precision Integer Types

The native data types in C++ are on 8-bit boundaries (8, 16, 32, and 64 bits). RTL signals and operations support arbitrary bit-lengths.

Vitis HLS provides arbitrary precision data types for C++ to allow variables and operations in the C++ code to be specified with any arbitrary bit-widths: 6-bit, 17-bit, 234-bit, up to 1024 bits.



TIP: The default maximum width allowed is 1024 bits. You can override this default by defining the macro `AP_INT_MAX_W` with a positive integer value less than or equal to 4096 before inclusion of the `ap_int.h` header file.

Arbitrary precision data types have two primary advantages over the native C++ types:

- Better quality hardware: If for example, a 17-bit multiplier is required, arbitrary precision types can specify that exactly 17-bit are used in the calculation.

Without arbitrary precision data types, such a multiplication (17-bit) must be implemented using 32-bit integer data types and result in the multiplication being implemented with multiple DSP modules.

- Accurate C++ simulation/analysis: Arbitrary precision data types in the C++ code allows the C++ simulation to be performed using accurate bit-widths and for the C++ simulation to validate the functionality (and accuracy) of the algorithm before synthesis.

The arbitrary precision types in C++ have none of the disadvantages of those in C:

- C++ arbitrary types can be compiled with standard C++ compilers (there is no C++ equivalent of `apcc`).
- C++ arbitrary precision types do not suffer from Integer Promotion Issues.

It is not uncommon for users to change a file extension from `.c` to `.cpp` so the file can be compiled as C++, where neither of these issues are present.

For the C++ language, the header file `ap_int.h` defines the arbitrary precision integer data types `ap_(u)int<W>`. For example, `ap_int<8>` represents an 8-bit signed integer data type and `ap_uint<234>` represents a 234-bit unsigned integer type.

The `ap_int.h` file is located in the directory `$HLS_ROOT/include`, where `$HLS_ROOT` is the Vitis HLS installation directory.

The code shown in the following example is a repeat of the code shown in the Basic Arithmetic example in [Standard Types](#). In this example, the data types in the top-level function to be synthesized are specified as `dinA_t`, `dinB_t`, and so on.

```
#include "cpp_ap_int_arith.h"

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

In this latest update to this example, the C++ arbitrary precision types are used:

- Add header file `ap_int.h` to the source code.
- Change the native C++ types to arbitrary precision types `ap_int<N>` or `ap_uint<N>`, where `N` is a bit-size from 1 to 1024 (as noted above, this can be extended to 4K-bits if required).

The data types are defined in the header `cpp_ap_int_arith.h`.

Compared with the Basic Arithmetic example in [Standard Types](#), the input data types have simply been reduced to represent the maximum size of the real input data (for example, 8-bit input `inA` is reduced to 6-bit input). The output types have been refined to be more accurate, for example, `out2`, the sum of `inA` and `inB`, need only be 13-bit and not 32-bit.

The following example shows basic arithmetic with C++ arbitrary precision types.

```
#ifndef _CPP_AP_INT_ARITH_H_
#define _CPP_AP_INT_ARITH_H_

#include <stdio.h>
#include "ap_int.h"

#define N 9

// Old data types
//typedef char dinA_t;
//typedef short dinB_t;
//typedef int dinC_t;
//typedef long long dinD_t;
//typedef int dout1_t;
```

```

//typedef unsigned int dout2_t;
//typedef int32_t dout3_t;
//typedef int64_t dout4_t;

typedef ap_int<6> dinA_t;
typedef ap_int<12> dinB_t;
typedef ap_int<22> dinC_t;
typedef ap_int<33> dinD_t;

typedef ap_int<18> dout1_t;
typedef ap_uint<13> dout2_t;
typedef ap_int<22> dout3_t;
typedef ap_int<6> dout4_t;

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

#endif

```

If [C++ Arbitrary Precision Integer Types](#) are synthesized, it results in a design that is functionally identical to [Standard Types](#). Rather than use the C++ `cout` operator to output the results to a file, the built-in `ap_int` method `.to_int()` is used to convert the `ap_int` results to integer types used with the standard `fprintf` function.

```

fprintf(fp, %d*%d=%d; %d+%d=%d; %d/%d=%d; %d mod %d=%d;\n,
       inA.to_int(), inB.to_int(), out1.to_int(),
       inB.to_int(), inA.to_int(), out2.to_int(),
       inC.to_int(), inA.to_int(), out3.to_int(),
       inD.to_int(), inA.to_int(), out4.to_int());

```

C++ Arbitrary Precision Integer Types: Reference Information

For comprehensive information on the methods, synthesis behavior, and all aspects of using the `ap_(u)int<N>` arbitrary precision data types, see [C++ Arbitrary Precision Types](#). This section includes:

- Techniques for assigning constant and initialization values to arbitrary precision integers (including values greater than 1024-bit).
- A description of Vitis HLS helper methods, such as printing, concatenating, bit-slicing and range selection functions.
- A description of operator behavior, including a description of shift operations (a negative shift value, results in a shift in the opposite direction).

C++ Arbitrary Precision Types

Vitis HLS provides a C++ template class, `ap_[u]int<>`, that implements arbitrary precision (or bit-accurate) integer data types with consistent, bit-accurate behavior between software and hardware modeling.

This class provides all arithmetic, bitwise, logical and relational operators allowed for native C integer types. In addition, this class provides methods to handle some useful hardware operations, such as allowing initialization and conversion of variables of widths greater than 64 bits. Details for all operators and class methods are discussed below.

Compiling ap_[u]int<> Types

To use the `ap_[u]int<>` classes, you must include the `ap_int.h` header file in all source files that reference `ap_[u]int<>` variables.

When compiling software models that use these classes, it may be necessary to specify the location of the Vitis HLS header files, for example by adding the `-I/<HLS_HOME>/include` option for `g++` compilation.

Declaring/Defining ap_[u] Variables

There are separate signed and unsigned classes:

- `ap_int<int_W>` (signed)
- `ap_uint<int_W>` (unsigned)

The template parameter `int_W` specifies the total width of the variable being declared.

User-defined types may be created with the C/C++ `typedef` statement as shown in the following examples:

```
include "ap_int.h">// use ap_[u]fixed<> types
typedef ap_uint<128> uint128_t; // 128-bit user defined type
ap_int<96> my_wide_var; // a global variable declaration
```

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



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

Following is an example of overriding `AP_INT_MAX_W`:

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

ap_int<4096> very_wide_var;
```

Initialization and Assignment from Constants (Literals)

The class constructor and assignment operator overloads, allows initialization of and assignment to `ap_[u]int<>` variables using standard C/C++ integer literals.

This method of assigning values to `ap_[u]int<>` variables is subject to the limitations of C++ and the system upon which the software will run. This typically leads to a 64-bit limit on integer literals (for example, for those `LL` or `ULL` suffixes).

To allow assignment of values wider than 64-bits, the `ap_[u]int<>` classes provide constructors that allow initialization from a string of arbitrary length (less than or equal to the width of the variable).

By default, the string provided is interpreted as a hexadecimal value as long as it contains only valid hexadecimal digits (that is, 0-9 and a-f). To assign a value from such a string, an explicit C++ style cast of the string to the appropriate type must be made.

Following are examples of initialization and assignments, including for values greater than 64-bit, are:

```
ap_int<42> a_42b_var(-1424692392255LL); // long long decimal format
a_42b_var = 0x14BB648B13FLL; // hexadecimal format

a_42b_var = -1; // negative int literal sign-extended to full width

ap_uint<96> wide_var("76543210fedcba9876543210", 16); // Greater than 64-bit
wide_var = ap_int<96>("0123456789abcdef01234567", 16);
```

Note: To avoid unexpected behavior during co-simulation, do not initialize `ap_uint<N> a = {0}`.

The `ap_[u]<>` constructor may be explicitly instructed to interpret the string as representing the number in radix 2, 8, 10, or 16 formats. This is accomplished by adding the appropriate radix value as a second parameter to the constructor call.

A compilation error occurs if the string literal contains any characters that are invalid as digits for the radix specified.

The following examples use different radix formats:

```
ap_int<6> a_6bit_var("101010", 2); // 42d in binary format
a_6bit_var = ap_int<6>("40", 8); // 32d in octal format
a_6bit_var = ap_int<6>("55", 10); // decimal format
a_6bit_var = ap_int<6>("2A", 16); // 42d in hexadecimal format

a_6bit_var = ap_int<6>("42", 2); // COMPILE-TIME ERROR! "42" is not binary
```

The radix of the number encoded in the string can also be inferred by the constructor, when it is prefixed with a zero (0) followed by one of the following characters: “b”, “o” or “x”. The prefixes “0b”, “0o” and “0x” correspond to binary, octal and hexadecimal formats respectively.

The following examples use alternate initializer string formats:

```
ap_int<6> a_6bit_var("0b101010", 2); // 42d in binary format
a_6bit_var = ap_int<6>("0o40", 8); // 32d in octal format
a_6bit_var = ap_int<6>("0x2A", 16); // 42d in hexidecimal format

a_6bit_var = ap_int<6>("0b42", 2); // COMPILE-TIME ERROR! "42" is not binary
```

If the bit-width is greater than 53-bits, the `ap_[u]int<>` value must be initialized with a string, for example:

```
ap_uint<72> Val("2460508560057040035.375");
```

Support for Console I/O (Printing)

As with initialization and assignment to `ap_[u]fixed<>` variables, Vitis HLS supports printing values that require more than 64-bits to represent.

Using the C++ Standard Output Stream

The easiest way to output any value stored in an `ap_[u]int` variable is to use the C++ standard output stream:

```
std::cout (#include <iostream> or <iostream.h>)
```

The stream insertion operator (`<<`) is overloaded to correctly output the full range of values possible for any given `ap_[u]fixed` variable. The following stream manipulators are also supported:

- `dec` (decimal)
- `hex` (hexadecimal)
- `oct` (octal)

These allow formatting of the value as indicated.

The following example uses `cout` to print values:

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_ufixed<72> Val("10fedcba9876543210");

cout << Val << endl; // Yields: "313512663723845890576"
cout << hex << val << endl; // Yields: "10fedcba9876543210"
cout << oct << val << endl; // Yields: "41773345651416625031020"
```

Using the Standard C Library

You can also use the standard C library (`#include <stdio.h>`) to print out values larger than 64-bits:

1. Convert the value to a C++ `std::string` using the `ap_[u]fixed` classes method `to_string()`.
2. Convert the result to a null-terminated C character string using the `std::string` class method `c_str()`.

Optional Argument One (Specifying the Radix)

You can pass the `ap[u]int::to_string()` method an optional argument specifying the radix of the numerical format desired. The valid radix argument values are:

- 2 (binary) (default)
- 8 (octal)
- 10 (decimal)
- 16 (hexadecimal)

Optional Argument Two (Printing as Signed Values)

A second optional argument to `ap_[u]int::to_string()` specifies whether to print the non-decimal formats as signed values. This argument is boolean. The default value is false, causing the non-decimal formats to be printed as unsigned values.

The following examples use `printf` to print values:

```
ap_int<72> Val("80fedcba9876543210");

printf("%s\n", Val.to_string().c_str()); // => "80FEDCBA9876543210"
printf("%s\n", Val.to_string(10).c_str()); // => "-2342818482890329542128"
printf("%s\n", Val.to_string(8).c_str()); // => "401773345651416625031020"
printf("%s\n", Val.to_string(16, true).c_str()); // => "-7F0123456789ABCDF0"
```

Expressions Involving `ap_[u]>` types

Variables of `ap_[u]>` types may generally be used freely in expressions involving C/C++ operators. Some behaviors may be unexpected. These are discussed in detail below.

Zero- and Sign-Extension on Assignment From Narrower to Wider Variables

When assigning the value of a narrower bit-width signed (`ap_int<>`) variable to a wider one, the value is sign-extended to the width of the destination variable, regardless of its signedness.

Similarly, an unsigned source variable is zero-extended before assignment.

Explicit casting of the source variable may be necessary to ensure expected behavior on assignment. See the following example:

```
ap_uint<10> Result;

ap_int<7> Val1 = 0x7f;
ap_uint<6> Val2 = 0x3f;

Result = Val1; // Yields: 0x3ff (sign-extended)
Result = Val2; // Yields: 0x03f (zero-padded)

Result = ap_uint<7>(Val1); // Yields: 0x07f (zero-padded)
Result = ap_int<6>(Val2); // Yields: 0x3ff (sign-extended)
```

Truncation on Assignment of Wider to Narrower Variables

Assigning the value of a wider source variable to a narrower one leads to truncation of the value. All bits beyond the most significant bit (MSB) position of the destination variable are lost.

There is no special handling of the sign information during truncation. This may lead to unexpected behavior. Explicit casting may help avoid this unexpected behavior.

Class Methods and Operators

The `ap_[u]int` types do not support implicit conversion from wide `ap_[u]int (>64bits)` to builtin C/C++ integer types. For example, the following code example return `s1`, because the implicit cast from `ap_int[65]` to `bool` in the if-statement returns a 0.

```
bool nonzero(ap_uint<65> data) {
    return data; // This leads to implicit truncation to 64b int
}

int main() {
    if (nonzero((ap_uint<65>)1 << 64)) {
        return 0;
    }
    printf(FAIL\n);
    return 1;
}
```

To convert wide `ap_[u]int` types to built-in integers, use the explicit conversion functions included with the `ap_[u]int` types:

- `to_int()`
- `to_long()`
- `to_bool()`

In general, any valid operation that can be done on a native C/C++ integer data type is supported using operator overloading for `ap_[u]int` types.

In addition to these overloaded operators, some class specific operators and methods are included to ease bit-level operations.

Binary Arithmetic Operators

Standard binary integer arithmetic operators are overloaded to provide arbitrary precision arithmetic. These operators take either:

- Two operands of `ap_[u]int`, or
- One `ap_[u]int` type and one C/C++ fundamental integer data type

For example:

- `char`

- short
- int

The width and signedness of the resulting value is determined by the width and signedness of the operands, before sign-extension, zero-padding or truncation are applied based on the width of the destination variable (or expression). Details of the return value are described for each operator.

When expressions contain a mix of `ap_[u]int` and C/C++ fundamental integer types, the C++ types assume the following widths:

- char (8-bits)
- short (16-bits)
- int (32-bits)
- long (32-bits)
- long long (64-bits)

Addition

```
ap_(u)int::RType ap_(u)int::operator + (ap_(u)int op)
```

Returns the sum of:

- Two `ap_[u]int`, or
- One `ap_[u]int` and a C/C++ integer type

The width of the sum value is:

- One bit more than the wider of the two operands, or
- Two bits if and only if the wider is unsigned and the narrower is signed

The sum is treated as signed if either (or both) of the operands is of a signed type.

Subtraction

```
ap_(u)int::RType ap_(u)int::operator - (ap_(u)int op)
```

Returns the difference of two integers.

The width of the difference value is:

- One bit more than the wider of the two operands, or
- Two bits if and only if the wider is unsigned and the narrower signed

This is true before assignment, at which point it is sign-extended, zero-padded, or truncated based on the width of the destination variable.

The difference is treated as signed regardless of the signedness of the operands.

Multiplication

```
ap_(u)int::RType ap_(u)int::operator * (ap_(u)int op)
```

Returns the product of two integer values.

The width of the product is the sum of the widths of the operands.

The product is treated as a signed type if either of the operands is of a signed type.

Division

```
ap_(u)int::RType ap_(u)int::operator / (ap_(u)int op)
```

Returns the quotient of two integer values.

The width of the quotient is the width of the dividend if the divisor is an unsigned type. Otherwise, it is the width of the dividend plus one.

The quotient is treated as a signed type if either of the operands is of a signed type.

Modulus

```
ap_(u)int::RType ap_(u)int::operator % (ap_(u)int op)
```

Returns the modulus, or remainder of integer division, for two integer values.

The width of the modulus is the minimum of the widths of the operands, if they are both of the same signedness.

If the divisor is an unsigned type and the dividend is signed, then the width is that of the divisor plus one.

The quotient is treated as having the same signedness as the dividend.



IMPORTANT! Vitis HLS synthesis of the modulus (%) operator will lead to instantiation of appropriately parameterized Xilinx LogiCORE divider cores in the generated RTL.

Following are examples of arithmetic operators:

```
ap_uint<71> Rslt;
ap_uint<42> Val1 = 5;
ap_int<23> Val2 = -8;
```

```
Rslt = Val1 + Val2; // Yields: -3 (43 bits) sign-extended to 71 bits
Rslt = Val1 - Val2; // Yields: +3 sign extended to 71 bits
Rslt = Val1 * Val2; // Yields: -40 (65 bits) sign extended to 71 bits
Rslt = 50 / Val2; // Yields: -6 (33 bits) sign extended to 71 bits
Rslt = 50 % Val2; // Yields: +2 (23 bits) sign extended to 71 bits
```

Bitwise Logical Operators

The bitwise logical operators all return a value with a width that is the maximum of the widths of the two operands. It is treated as unsigned if and only if both operands are unsigned. Otherwise, it is of a signed type.

Sign-extension (or zero-padding) may occur, based on the signedness of the expression, not the destination variable.

Bitwise OR

```
ap_(u)int::RType ap_(u)int::operator | (ap_(u)int op)
```

Returns the bitwise OR of the two operands.

Bitwise AND

```
ap_(u)int::RType ap_(u)int::operator & (ap_(u)int op)
```

Returns the bitwise AND of the two operands.

Bitwise XOR

```
ap_(u)int::RType ap_(u)int::operator ^ (ap_(u)int op)
```

Returns the bitwise XOR of the two operands.

Unary Operators

Addition

```
ap_(u)int ap_(u)int::operator + ()
```

Returns the self copy of the `ap_[u]int` operand.

Subtraction

```
ap_(u)int::RType ap_(u)int::operator - ()
```

Returns the following:

- The negated value of the operand with the same width if it is a signed type, or
- Its width plus one if it is unsigned.

The return value is always a signed type.

Bitwise Inverse

```
ap_(u)int::RType ap_(u)int::operator ~ ()
```

Returns the bitwise-NOT of the operand with the same width and signedness.

Logical Invert

```
bool ap_(u)int::operator ! ()
```

Returns a Boolean `false` value if and only if the operand is *not* equal to zero (0).

Returns a Boolean `true` value if the operand is equal to zero (0).

Ternary Operators

When you use the ternary operator with the standard C `int` type, you must explicitly cast from one type to the other to ensure that both results have the same type. For example:

```
// Integer type is cast to ap_int type
ap_int<32> testc3(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?ap_int<32>(a):b;
}
// ap_int type is cast to an integer type
ap_int<32> testc4(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?a+1:(int)b;
}
// Integer type is cast to ap_int type
ap_int<32> testc5(int a, ap_int<32> b, ap_int<32> c, bool d) {
    return d?ap_int<32>(a):b+1;
}
```

Shift Operators

Each shift operator comes in two versions:

- One version for *unsigned* right-hand side (RHS) operands
- One version for *signed* right-hand side (RHS) operands

A negative value supplied to the signed RHS versions reverses the shift operations direction. That is, a shift by the absolute value of the RHS operand in the opposite direction occurs.

The shift operators return a value with the same width as the left-hand side (LHS) operand. As with C/C++, if the LHS operand of a shift-right is a signed type, the sign bit is copied into the most significant bit positions, maintaining the sign of the LHS operand.

Unsigned Integer Shift Right

```
ap_(u)int ap_(u)int::operator >> (ap_uint<int_W2> op)
```

Integer Shift Right

```
ap_(u)int ap_(u)int::operator >> (ap_int<int_W2> op)
```

Unsigned Integer Shift Left

```
ap_(u)int ap_(u)int::operator << (ap_uint<int_W2> op)
```

Integer Shift Left

```
ap_(u)int ap_(u)int::operator << (ap_int<int_W2> op)
```



CAUTION! When assigning the result of a shift-left operator to a wider destination variable, some or all information may be lost. Xilinx recommends that you explicitly cast the shift expression to the destination type to avoid unexpected behavior.

Following are examples of shift operations:

```
ap_uint<13> Rslt;  
  
ap_uint<7> Val1 = 0x41;  
  
Rslt = Val1 << 6; // Yields: 0x0040, i.e. msb of Val1 is lost  
Rslt = ap_uint<13>(Val1) << 6; // Yields: 0x1040, no info lost  
  
ap_int<7> Val2 = -63;  
Rslt = Val2 >> 4; //Yields: 0x1ffc, sign is maintained and extended
```

Compound Assignment Operators

Vitis HLS supports compound assignment operators:

```
*=  
/=  
%=  
+=  
-=  
<<=  
>>=  
&=  
^=  
|=
```

The RHS expression is first evaluated then supplied as the RHS operand to the base operator, the result of which is assigned back to the LHS variable. The expression sizing, signedness, and potential sign-extension or truncation rules apply as discussed above for the relevant operations.

```
ap_uint<10> Val1 = 630;  
ap_int<3> Val2 = -3;  
ap_uint<5> Val3 = 27;  
  
Val1 += Val2 - Val3; // Yields: 600 and is equivalent to:  
  
// Val1 = ap_uint<10>(ap_int<11>(Val1) +  
// ap_int<11>((ap_int<6>(Val2) -  
// ap_int<6>(Val3))));
```

Increment and Decrement Operators

The increment and decrement operators are provided. All return a value of the same width as the operand and which is unsigned if and only if both operands are of unsigned types and signed otherwise.

Pre-Increment

```
ap_(u)int& ap_(u)int::operator ++ ()
```

Returns the incremented value of the operand.

Assigns the incremented value to the operand.

Post-Increment

```
const ap_(u)int ap_(u)int::operator ++ (int)
```

Returns the value of the operand before assignment of the incremented value to the operand variable.

Pre-Decrement

```
ap_(u)int& ap_(u)int::operator -- ()
```

Returns the decremented value of, as well as assigning the decremented value to, the operand.

Post-Decrement

```
const ap_(u)int ap_(u)int::operator -- (int)
```

Returns the value of the operand before assignment of the decremented value to the operand variable.

Relational Operators

Vitis HLS supports all relational operators. They return a Boolean value based on the result of the comparison. You can compare variables of `ap_[u]int` types to C/C++ fundamental integer types with these operators.

Equality

```
bool ap_(u)int::operator == (ap_(u)int op)
```

Inequality

```
bool ap_(u)int::operator != (ap_(u)int op)
```

Less than

```
bool ap_(u)int::operator < (ap_(u)int op)
```

Greater than

```
bool ap_(u)int::operator > (ap_(u)int op)
```

Less than or equal to

```
bool ap_(u)int::operator <= (ap_(u)int op)
```

Greater than or equal to

```
bool ap_(u)int::operator >= (ap_(u)int op)
```

Other Class Methods, Operators, and Data Members

The following sections discuss other class methods, operators, and data members.

Bit-Level Operations

The following methods facilitate common bit-level operations on the value stored in `ap_[u]int` type variables.

Length

```
int ap_(u)int::length ()
```

Returns an integer value providing the total number of bits in the `ap_[u]int` variable.

Concatenation

```
ap_concat_ref ap_(u)int::concat (ap_(u)int low)
ap_concat_ref ap_(u)int::operator , (ap_(u)int high, ap_(u)int low)
```

Concatenates two `ap_[u]int` variables, the width of the returned value is the sum of the widths of the operands.

The High and Low arguments are placed in the higher and lower order bits of the result respectively; the `concat()` method places the argument in the lower order bits.

When using the overloaded comma operator, the parentheses are required. The comma operator version may also appear on the LHS of assignment.



RECOMMENDED: To avoid unexpected results, explicitly cast C/C++ native types (including integer literals) to an appropriate `ap_[u]int` type before concatenating.

```
ap_uint<10> Rslt;
ap_int<3> Val1 = -3;
ap_int<7> Val2 = 54;

Rslt = (Val2, Val1); // Yields: 0x1B5
Rslt = Val1.concat(Val2); // Yields: 0x2B6
(Val1, Val2) = 0xAB; // Yields: Val1 == 1, Val2 == 43
```

Bit Selection

```
ap_bit_ref ap_(u)int::operator [] (int bit)
```

Selects one bit from an arbitrary precision integer value and returns it.

The returned value is a reference value that can set or clear the corresponding bit in this `ap_[u]int`.

The bit argument must be an `int` value. It specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `ap_[u]int`.

The result type `ap_bit_ref` represents the reference to one bit of this `ap_[u]int` instance specified by bit.

Range Selection

```
ap_range_ref ap_(u)int::range (unsigned Hi, unsigned Lo)
ap_range_ref ap_(u)int::operator () (unsigned Hi, unsigned Lo)
```

Returns the value represented by the range of bits specified by the arguments.

The `Hi` argument specifies the most significant bit (MSB) position of the range, and `Lo` specifies the least significant bit (LSB).

The LSB of the source variable is in position 0. If the `Hi` argument has a value less than `Lo`, the bits are returned in reverse order.

```
ap_uint<4> Rslt;
ap_uint<8> Val1 = 0x5f;
ap_uint<8> Val2 = 0xaa;

Rslt = Val1.range(3, 0); // Yields: 0xF
Val1(3,0) = Val2(3, 0); // Yields: 0x5A
Val1(3,0) = Val2(4, 1); // Yields: 0x55
Rslt = Val1.range(4, 7); // Yields: 0xA; bit-reversed!
```

Note: The object returned by range select is not an `ap_(u)int` object and lacks operators, but can be used for assignment. To use the range select result in a chained expression with `ap_(u)int` methods, add an explicit constructor like below.

```
ap_uint<32> v = 0x8fff0000;
bool r = ap_uint<16>(v.range(23, 8)).xor_reduce();
```

AND reduce

```
bool ap_(u)int::and_reduce()
```

- Applies the AND operation on all bits in this `ap_(u)int`.
- Returns the resulting single bit.
- Equivalent to comparing this value against -1 (all ones) and returning `true` if it matches, `false` otherwise.

OR reduce

```
bool ap_(u)int::or_reduce()
```

- Applies the OR operation on all bits in this `ap_(u)int`.
- Returns the resulting single bit.
- Equivalent to comparing this value against 0 (all zeros) and returning `false` if it matches, `true` otherwise.

XOR reduce

```
bool ap_(u)int::xor_reduce()
```

- Applies the XOR operation on all bits in this `ap_int`.
- Returns the resulting single bit.
- Equivalent to counting the number of 1 bits in this value and returning `false` if the count is even or `true` if the count is odd.

NAND reduce

```
bool ap_(u)int::nand_reduce ()
```

- Applies the NAND operation on all bits in this `ap_int`.
- Returns the resulting single bit.
- Equivalent to comparing this value against -1 (all ones) and returning `false` if it matches, `true` otherwise.

NOR reduce

```
bool ap_int::nor_reduce ()
```

- Applies the NOR operation on all bits in this `ap_int`.
- Returns the resulting single bit.
- Equivalent to comparing this value against 0 (all zeros) and returning `true` if it matches, `false` otherwise.

XNOR reduce

```
bool ap_(u)int::xnor_reduce ()
```

- Applies the XNOR operation on all bits in this `ap_(u)int`.
- Returns the resulting single bit.
- Equivalent to counting the number of 1 bits in this value and returning `true` if the count is even or `false` if the count is odd.

Bit Reduction Method Examples

```
ap_uint<8> Val = 0xaa;  
  
bool t = Val.and_reduce(); // Yields: false  
t = Val.or_reduce(); // Yields: true  
t = Val.xor_reduce(); // Yields: false  
t = Val.nand_reduce(); // Yields: true  
t = Val.nor_reduce(); // Yields: false  
t = Val.xnor_reduce(); // Yields: true
```

Bit Reverse

```
void ap_(u)int::reverse ()
```

Reverses the contents of `ap_[u]int` instance:

- The LSB becomes the MSB.
- The MSB becomes the LSB.

Reverse Method Example

```
ap_uint<8> Val = 0x12;  
Val.reverse(); // Yields: 0x48
```

Test Bit Value

```
bool ap_(u)int::test (unsigned i)
```

Checks whether specified bit of `ap_(u)int` instance is 1.

Returns true if Yes, false if No.

Test Method Example

```
ap_uint<8> Val = 0x12;  
bool t = Val.test(5); // Yields: true
```

Set Bit Value

```
void ap_(u)int::set (unsigned i, bool v)  
void ap_(u)int::set_bit (unsigned i, bool v)
```

Sets the specified bit of the `ap_(u)int` instance to the value of integer `v`.

Set Bit (to 1)

```
void ap_(u)int::set (unsigned i)
```

Sets the specified bit of the `ap_(u)int` instance to the value 1 (one).

Clear Bit (to 0)

```
void ap_(u)int:: clear(unsigned i)
```

Sets the specified bit of the `ap_(u)int` instance to the value 0 (zero).

Invert Bit

```
void ap_(u)int:: invert(unsigned i)
```

Inverts the bit specified in the function argument of the `ap_(u)int` instance. The specified bit becomes 0 if its original value is 1 and vice versa.

Example of bit set, clear and invert bit methods:

```
ap_uint<8> Val = 0x12;
Val.set(0, 1); // Yields: 0x13
Val.set_bit(4, false); // Yields: 0x03
Val.set(7); // Yields: 0x83
Val.clear(1); // Yields: 0x81
Val.invert(4); // Yields: 0x91
```

Rotate Right

```
void ap_(u)int:: rrotate(unsigned n)
```

Rotates the `ap_(u)int` instance `n` places to right.

Rotate Left

```
void ap_(u)int:: lrotate(unsigned n)
```

Rotates the `ap_(u)int` instance `n` places to left.

```
ap_uint<8> Val = 0x12;

Val.rrotate(3); // Yields: 0x42
Val.lrotate(6); // Yields: 0x90
```

Bitwise NOT

```
void ap_(u)int:: b_not()
```

- Complements every bit of the `ap_(u)int` instance.

```
ap_uint<8> Val = 0x12;

Val.b_not(); // Yields: 0xED
```

Bitwise NOT Example

Test Sign

```
bool ap_int:: sign()
```

- Checks whether the `ap_(u)int` instance is negative.
- Returns `true` if negative.
- Returns `false` if positive.

Explicit Conversion Methods

To C/C++ “(u)int”

```
int ap_(u)int::to_int ()
unsigned ap_(u)int::to_uint ()
```

- Returns native C/C++ (32-bit on most systems) integers with the value contained in the `ap_[u]int`.
- Truncation occurs if the value is greater than can be represented by an `[unsigned]` `int`.

To C/C++ 64-bit “(u)int”

```
long long ap_(u)int::to_int64 ()
unsigned long long ap_(u)int::to_uint64 ()
```

- Returns native C/C++ 64-bit integers with the value contained in the `ap_[u]int`.
- Truncation occurs if the value is greater than can be represented by an `[unsigned]` `int`.

To C/C++ “double”

```
double ap_(u)int::to_double ()
```

- Returns a native C/C++ `double` 64-bit floating point representation of the value contained in the `ap_[u]int`.
- If the `ap_[u]int` is wider than 53 bits (the number of bits in the mantissa of a `double`), the resulting `double` may not have the exact value expected.



RECOMMENDED: Xilinx recommends that you explicitly call member functions instead of using C-style cast to convert `ap_[u]int` to other data types.

Sizeof

The standard C++ `sizeof()` function should not be used with `ap_[u]int` or other classes or instance of object. The `ap_int<>` data type is a class and `sizeof` returns the storage used by that class or instance object. `sizeof(ap_int<N>)` always returns the number of bytes used. For example:

```
sizeof(ap_int<127>)=16
sizeof(ap_int<128>)=16
sizeof(ap_int<129>)=24
sizeof(ap_int<130>)=24
```

Compile Time Access to Data Type Attributes

The `ap_[u]int<>` types are provided with a static member that allows the size of the variables to be determined at compile time. The data type is provided with the static const member `width`, which is automatically assigned the width of the data type:

```
static const int width = _AP_W;
```

You can use the `width` data member to extract the data width of an existing `ap_[u]int<>` data type to create another `ap_[u]int<>` data type at compile time. The following example shows how the size of variable `Res` is defined as 1-bit greater than variables `Val1` and `Val2`:

```
// Definition of basic data type
#define INPUT_DATA_WIDTH 8
typedef ap_int<INPUT_DATA_WIDTH> data_t;
// Definition of variables
data_t Val1, Val2;
// Res is automatically sized at compile-time to be 1-bit greater than data
// type
data_t
ap_int<data_t::width+1> Res = Val1 + Val2;
```

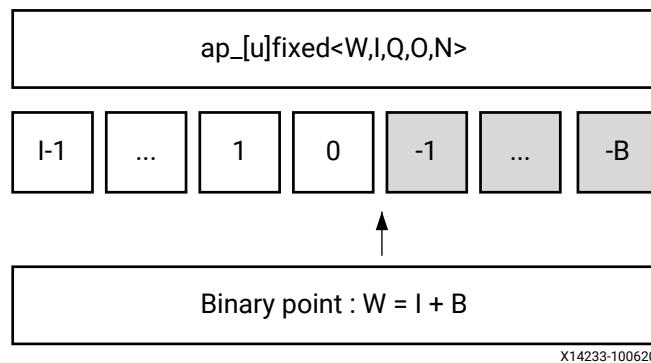
This ensures that Vitis HLS correctly models the bit-growth caused by the addition even if you update the value of `INPUT_DATA_WIDTH` for `data_t`.

C++ Arbitrary Precision Fixed-Point Types

C++ functions can take advantage of the arbitrary precision fixed-point types included with Vitis HLS. The following figure summarizes the basic features of these fixed-point types:

- The word can be signed (`ap_fixed`) or unsigned (`ap_ufixed`).
- A word with of any arbitrary size `W` can be defined.
- The number of places above the decimal point `I`, also defines the number of decimal places in the word, `W-I` (represented by `B` in the following figure).
- The type of rounding or quantization (`Q`) can be selected.
- The overflow behavior (`O` and `N`) can be selected.

Figure 127: Arbitrary Precision Fixed-Point Types



TIP: The arbitrary precision fixed-point types can be used when header file `ap_fixed.h` is included in the code.

Arbitrary precision fixed-point types use more memory during C simulation. If using very large arrays of `ap_[u]fixed` types, refer to the discussion of C simulation in [Arrays](#).

The advantages of using fixed-point types are:

- They allow fractional number to be easily represented.
- When variables have a different number of integer and decimal place bits, the alignment of the decimal point is handled.
- There are numerous options to handle how rounding should happen: when there are too few decimal bits to represent the precision of the result.
- There are numerous options to handle how variables should overflow: when the result is greater than the number of integer bits can represent.

These attributes are summarized by examining the code in the example below. First, the header file `ap_fixed.h` is included. The `ap_fixed` types are then defined using the `typedef` statement:

- A 10-bit input: 8-bit integer value with 2 decimal places.
- A 6-bit input: 3-bit integer value with 3 decimal places.
- A 22-bit variable for the accumulation: 17-bit integer value with 5 decimal places.
- A 36-bit variable for the result: 30-bit integer value with 6 decimal places.

The function contains no code to manage the alignment of the decimal point after operations are performed. The alignment is done automatically.

The following code sample shows ap_fixed type.

```
#include "ap_fixed.h"

typedef ap_ufixed<10, 8, AP_RND, AP_SAT> din1_t;
typedef ap_fixed<6, 3, AP_RND, AP_WRAP> din2_t;
typedef ap_fixed<22,17, AP_TRN, AP_SAT> dint_t;
typedef ap_fixed<36,30> dout_t;

dout_t cpp_ap_fixed(din1_t d_in1, din2_t d_in2) {
    static dint_t sum;
    sum += d_in1;
    return sum * d_in2;
}
```

Using ap_(u)fixed types, the C++ simulation is bit accurate. Fast simulation can validate the algorithm and its accuracy. After synthesis, the RTL exhibits the identical bit-accurate behavior.

Arbitrary precision fixed-point types can be freely assigned literal values in the code. This is shown in the test bench (see the example below) used with the example above, in which the values of `in1` and `in2` are declared and assigned constant values.

When assigning literal values involving operators, the literal values must first be cast to ap_(u)fixed types. Otherwise, the C compiler and Vitis HLS interpret the literal as an integer or float/double type and may fail to find a suitable operator. As shown in the following example, in the assignment of `in1 = in1 + din1_t(0.25)`, the literal 0.25 is cast to an ap_fixed type.

```
#include <cmath>
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;
#include "ap_fixed.h"

typedef ap_ufixed<10, 8, AP_RND, AP_SAT> din1_t;
typedef ap_fixed<6, 3, AP_RND, AP_WRAP> din2_t;
typedef ap_fixed<22,17, AP_TRN, AP_SAT> dint_t;
typedef ap_fixed<36,30> dout_t;

dout_t cpp_ap_fixed(din1_t d_in1, din2_t d_in2);
int main()
{
    ofstream result;
    din1_t in1 = 0.25;
    din2_t in2 = 2.125;
    dout_t output;
    int retval=0;

    result.open(result.dat);
    // Persistent manipulators
    result << right << fixed << setbase(10) << setprecision(15);

    for (int i = 0; i <= 250; i++)
    {
```

```

        output = cpp_ap_fixed(in1,in2);

result << setw(10) << i;
result << setw(20) << in1;
result << setw(20) << in2;
result << setw(20) << output;
result << endl;

in1 = in1 + din1_t(0.25);
in2 = in2 - din2_t(0.125);
}
result.close();

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

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

```

Fixed-Point Identifier Summary

The following table shows the quantization and overflow modes.



TIP: Quantization and overflow modes that do more than the default behavior of standard hardware arithmetic (wrap and truncate) result in operators with more associated hardware. It costs logic (LUTs) to implement the more advanced modes, such as round to minus infinity or saturate symmetrically.

Table 41: Fixed-Point Identifier Summary

Identifier	Description
W	Word length in bits
I	<p>The number of bits used to represent the integer value, that is, the number of integer bits to the <i>left</i> of the binary point. When this value is negative, it represents the number of <i>implicit</i> sign bits (for signed representation), or the number of <i>implicit</i> zero bits (for unsigned representation) to the <i>right</i> of the binary point. For example,</p> <pre> ap_fixed<2, 0> a = -0.5; // a can be -0.5, ap_ufixed<1, 0> x = 0.5; // 1-bit representation. x can be 0 or 0.5 ap_ufixed<1, -1> y = 0.25; // 1-bit representation. y can be 0 or 0.25 const ap_fixed<1, -7> z = 1.0/256; // 1-bit representation for z = 2^-8 </pre>

Table 41: Fixed-Point Identifier Summary (cont'd)

Identifier	Description	
Q	Quantization mode: This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result.	
	ap_fixed Types	Description
	AP_RND	Round to plus infinity
	AP_RND_ZERO	Round to zero
	AP_RND_MIN_INF	Round to minus infinity
	AP_RND_INF	Round to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity (default)
	AP_TRN_ZERO	Truncation to zero
O	Overflow mode: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result.	
	ap_fixed Types	Description
	AP_SAT ¹	Saturation
	AP_SAT_ZERO ¹	Saturation to zero
	AP_SAT_SYM ¹	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	This defines the number of saturation bits in overflow wrap modes.	

Notes:

1. Using the AP_SAT* modes can result in higher resource usage as extra logic will be needed to perform saturation and this extra cost can be as high as 20% additional LUT usage.

C++ Arbitrary Precision Fixed-Point Types: Reference Information

For comprehensive information on the methods, synthesis behavior, and all aspects of using the `ap_(u)fixed<N>` arbitrary precision fixed-point data types, see [C++ Arbitrary Precision Fixed-Point Types](#). This section includes:

- Techniques for assigning constant and initialization values to arbitrary precision integers (including values greater than 1024-bit).
- A detailed description of the overflow and saturation modes.
- A description of Vitis HLS helper methods, such as printing, concatenating, bit-slicing and range selection functions.
- A description of operator behavior, including a description of shift operations (a negative shift values, results in a shift in the opposite direction).



IMPORTANT! For the compiler to process, you must use the appropriate header files for the language.

C++ Arbitrary Precision Fixed-Point Types

Vitis HLS supports fixed-point types that allow fractional arithmetic to be easily handled. The advantage of fixed-point arithmetic is shown in the following example.

```
ap_fixed<11, 6> Var1 = 22.96875; // 11-bit signed word, 5 fractional bits
ap_ufixed<12,11> Var2 = 512.5; // 12-bit word, 1 fractional bit
ap_fixed<16,11> Res1; // 16-bit signed word, 5 fractional bits

Res1 = Var1 + Var2; // Result is 535.46875
```

Even though `Var1` and `Var2` have different precisions, the fixed-point type ensures that the decimal point is correctly aligned before the operation (an addition in this case), is performed. You are not required to perform any operations in the C code to align the decimal point.

The type used to store the result of any fixed-point arithmetic operation must be large enough (in both the integer and fractional bits) to store the full result.

If this is not the case, the `ap_fixed` type performs:

- overflow handling (when the result has more MSBs than the assigned type supports)
- quantization (or rounding, when the result has fewer LSBs than the assigned type supports)

The `ap_[u]fixed` type provides various options on how the overflow and quantization are performed. The options are discussed below.

ap_[u]fixed Representation

In `ap[u]fixed` types, a fixed-point value is represented as a sequence of bits with a specified position for the binary point.

- Bits to the left of the binary point represent the integer part of the value.
- Bits to the right of the binary point represent the fractional part of the value.

`ap_[u]fixed` type is defined as follows:

```
ap_[u]fixed<int W,
           int I,
           ap_q_mode Q,
           ap_o_mode O,
           ap_sat_bits N>;
```

Quantization Modes

Rounding to plus infinity	AP_RND
Rounding to zero	AP_RND_ZERO
Rounding to minus infinity	AP_RND_MIN_INF
Rounding to infinity	AP_RND_INF
Convergent rounding	AP_RND_CONV

Truncation
 Truncation to zero

[AP_TRN](#)
[AP_TRN_ZERO](#)

AP_RND

- Round the value to the nearest representable value for the specific `ap_[u]fixed` type.

```
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.5
ap_fixed<3, 2, AP_RND, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0
```

AP_RND_ZERO

- Round the value to the nearest representable value.
- Round towards zero.
 - For positive values, delete the redundant bits.
 - For negative values, add the least significant bits to get the nearest representable value.

```
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_RND_ZERO, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0
```

AP_RND_MIN_INF

- Round the value to the nearest representable value.
- Round towards minus infinity.
 - For positive values, delete the redundant bits.
 - For negative values, add the least significant bits.

```
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_RND_MIN_INF, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.5
```

AP_RND_INF

- Round the value to the nearest representable value.
- The rounding depends on the least significant bit.
 - For positive values, if the least significant bit is set, round towards plus infinity. Otherwise, round towards minus infinity.
 - For negative values, if the least significant bit is set, round towards minus infinity. Otherwise, round towards plus infinity.

```
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.5
ap_fixed<3, 2, AP_RND_INF, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.5
```

AP_RND_CONV

- Round to the nearest representable value with "ties" rounding to even, that is, the least significant bit (after rounding) is forced to zero.
- A "tie" is the midpoint of two representable values and occurs when the bit following the least significant bit (after rounding) is 1 and all the bits below it are zero.

```
// For the following examples, bit3 of the 8-bit value becomes the
// LSB of the final 5-bit value (after rounding).
// Notes:
//   * bit7 of the 8-bit value is the MSB (sign bit)
//   * the 3 LSBs of the 8-bit value (bit2, bit1, bit0) are treated as
//     guard, round and sticky bits.
//   * See http://pages.cs.wisc.edu/~david/courses/cs552/S12/handouts/
//     guardbits.pdf

ap_fixed<8,3> p1 = 1.59375; // p1 = 001.10011
ap_fixed<5,3,AP_RND_CONV> rconv1 = p1; // rconv1 = 1.5 (001.10)

ap_fixed<8,3> p2 = 1.625; // p2 = 001.10100 => tie with bit3 (LSB-to-be)
= 0
ap_fixed<5,3,AP_RND_CONV> rconv2 = p2; // rconv2 = 1.5 (001.10) => lsb is
already zero, just truncate

ap_fixed<8,3> p3 = 1.375; // p3 = 001.01100 => tie with bit3 (LSB-to-be)
= 1
ap_fixed<5,3,AP_RND_CONV> rconv3 = p3; // rconv3 = 1.5 (001.10) => lsb is
made zero by rounding up

ap_fixed<8,3> p3 = 1.65625; // p3 = 001.10101
ap_fixed<5,3,AP_RND_CONV> rconv3 = p3; // rconv3 = 1.75 (001.11) => round
up
```

AP_TRN

- Always round the value towards minus infinity.

```
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_TRN, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.5
```

AP_TRN_ZERO

Round the value to:

- For positive values, the rounding is the same as mode AP_TRN.
- For negative values, round towards zero.

```
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = 1.25; // Yields: 1.0
ap_fixed<3, 2, AP_TRN_ZERO, AP_SAT> UAPFixed4 = -1.25; // Yields: -1.0
```

Overflow Modes

Saturation
 Saturation to zero

[AP_SAT](#)
[AP_SAT_ZERO](#)

Symmetrical saturation	AP_SAT_SYM
Wrap-around	AP_WRAP
Sign magnitude wrap-around	AP_WRAP_SM

AP_SAT

Saturate the value.

- To the maximum value in case of overflow.
- To the negative maximum value in case of negative overflow.

```
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields: 7.0
ap_fixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields: -8.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = 19.0; // Yields: 15.0
ap_ufixed<4, 4, AP_RND, AP_SAT> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_SAT_ZERO

Force the value to zero in case of overflow, or negative overflow.

```
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0; // Yields: 0.0
ap_fixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields: 0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = 19.0; // Yields: 0.0
ap_ufixed<4, 4, AP_RND, AP_SAT_ZERO> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_SAT_SYM

Saturate the value:

- To the maximum value in case of overflow.
- To the minimum value in case of negative overflow.
 - Negative maximum for signed `ap_fixed` types
 - Zero for unsigned `ap_ufixed` types

```
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields: 7.0
ap_fixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields: -7.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = 19.0; // Yields: 15.0
ap_ufixed<4, 4, AP_RND, AP_SAT_SYM> UAPFixed4 = -19.0; // Yields: 0.0
```

AP_WRAP

Wrap the value around in case of overflow.

```
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 31.0; // Yields: -1.0
ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields: -3.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 19.0; // Yields: 3.0
ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; // Yields: 13.0
```

If the value of N is set to zero (the default overflow mode):

- All MSB bits outside the range are deleted.
- For unsigned numbers. After the maximum it wraps around to zero.
- For signed numbers. After the maximum, it wraps to the minimum values.

If N>0:

- When N > 0, N MSB bits are saturated or set to 1.
- The sign bit is retained, so positive numbers remain positive and negative numbers remain negative.
- The bits that are not saturated are copied starting from the LSB side.

AP_WRAP_SM

The value should be sign-magnitude wrapped around.

```
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = 19.0; // Yields: -4.0
ap_fixed<4, 4, AP_RND, AP_WRAP_SM> UAPFixed4 = -19.0; // Yields: 2.0
```

If the value of N is set to zero (the default overflow mode):

- This mode uses sign magnitude wrapping.
- Sign bit set to the value of the least significant deleted bit.
- If the most significant remaining bit is different from the original MSB, all the remaining bits are inverted.
- If MSBs are same, the other bits are copied over.
 1. Delete redundant MSBs.
 2. The new sign bit is the least significant bit of the deleted bits. 0 in this case.
 3. Compare the new sign bit with the sign of the new value.
- If different, invert all the numbers. They are different in this case.

If N>0:

- Uses sign magnitude saturation
- N MSBs are saturated to 1.
- Behaves similar to a case in which N = 0, except that positive numbers stay positive and negative numbers stay negative.

Compiling ap_[u]fixed<> Types

To use the `ap_[u]fixed<>` classes, you must include the `ap_fixed.h` header file in all source files that reference `ap_[u]fixed<>` variables.

When compiling software models that use these classes, it may be necessary to specify the location of the Vitis HLS header files, for example by adding the “`-I/<HLS_HOME>/include`” option for `g++` compilation.

Declaring and Defining `ap_[u]fixed<>` Variables

There are separate signed and unsigned classes:

- `ap_fixed<W, I>` (signed)
- `ap_ufixed<W, I>` (unsigned)

You can create user-defined types with the C/C++ `typedef` statement:

```
#include "ap_fixed.h" // use ap_[u]fixed<> types
typedef ap_ufixed<128, 32> uint128_t; // 128-bit user defined type,
// 32 integer bits
```

Initialization and Assignment from Constants (Literals)

You can initialize `ap_[u]fixed` variable with normal floating point constants of the usual C/C++ width:

- 32 bits for type `float`
- 64 bits for type `double`

That is, typically, a floating point value that is single precision type or in the form of double precision.

Note that the value assigned to the fixed-point variable will be limited by the precision of the constant. Use string initialization as described in [Initialization and Assignment from Constants \(Literals\)](#) to ensure that all bits of the fixed-point variable are populated according to the precision described by the string.

```
#include <ap_fixed.h>

ap_ufixed<30, 15> my15BitInt = 3.1415;
ap_fixed<42, 23> my42BitInt = -1158.987;
ap_ufixed<99, 40> = 287432.0382911;
ap_fixed<36, 30> = -0x123.456p-1;
```

The `ap_[u]fixed` types do not support initialization if they are used in an array of `std::complex` types.

```
typedef ap_fixed<DIN_W, 1, AP_TRN, AP_SAT> coeff_t; // MUST have IW >= 1
std::complex<coeff_t> twid_rom[REAL_SZ/2] = {{ 1, -0 }, { 0.9, -0.006 }, etc.}
```

The initialization values must first be cast to `std::complex`:

```
typedef ap_fixed<DIN_W, 1, AP_TRN, AP_SAT> coeff_t; // MUST have IW >= 1
std::complex<coeff_t> twid_rom[REAL_SZ/2] = {std::complex<coeff_t>( 1,
-0 ),
std::complex<coeff_t>(0.9,-0.006 ),etc.}
```

Support for Console I/O (Printing)

As with initialization and assignment to `ap_[u]fixed<>` variables, Vitis HLS supports printing values that require more than 64 bits to represent.

The easiest way to output any value stored in an `ap_[u]fixed` variable is to use the C++ standard output stream, `std::cout` (`#include <iostream>` or `<iostream.h>`). The stream insertion operator, “`<<`”, is overloaded to correctly output the full range of values possible for any given `ap_[u]fixed` variable. The following stream manipulators are also supported, allowing formatting of the value as shown.

- `dec` (decimal)
- `hex` (hexadecimal)
- `oct` (octal)

```
#include <iostream.h>
// Alternative: #include <iostream>

ap_fixed<6,3, AP_RND, AP_WRAP> Val = 3.25;

cout << Val << endl;      // Yields: 3.25
```

Using the Standard C Library

You can also use the standard C library (`#include <stdio.h>`) to print out values larger than 64-bits:

1. Convert the value to a C++ `std::string` using the `ap_[u]fixed` classes method `to_string()`.
2. Convert the result to a null-terminated C character string using the `std::string` class method `c_str()`.

Optional Argument One (Specifying the Radix)

You can pass the `ap[u]int::to_string()` method an optional argument specifying the radix of the numerical format desired. The valid radix argument values are:

- 2 (binary)
- 8 (octal)
- 10 (decimal)

- 16 (hexadecimal) (default)

Optional Argument Two (Printing as Signed Values)

A second optional argument to `ap_[u]int::to_string()` specifies whether to print the non-decimal formats as signed values. This argument is boolean. The default value is false, causing the non-decimal formats to be printed as unsigned values.

```
ap_fixed<6,3, AP_RND, AP_WRAP> Val = 3.25;  
  
printf("%s \n", in2.to_string().c_str()); // Yields: 0b011.010  
printf("%s \n", in2.to_string(10).c_str()); //Yields: 3.25
```

The `ap_[u]fixed` types are supported by the following C++ manipulator functions:

- `setprecision`
- `setw`
- `setfill`

The `setprecision` manipulator sets the decimal precision to be used. It takes one parameter `f` as the value of decimal precision, where `n` specifies the maximum number of meaningful digits to display in total (counting both those before and those after the decimal point).

The default value of `f` is 6, which is consistent with native C float type.

```
ap_fixed<64, 32> f = 3.14159;  
cout << setprecision (5) << f << endl;  
cout << setprecision (9) << f << endl;  
f = 123456;  
cout << setprecision (5) << f << endl;
```

The example above displays the following results where the printed results are rounded when the actual precision exceeds the specified precision:

```
3.1416  
3.14159  
1.2346e+05
```

The `setw` manipulator:

- Sets the number of characters to be used for the field width.
- Takes one parameter `w` as the value of the width

where

- `w` determines the minimum number of characters to be written in some output representation.

If the standard width of the representation is shorter than the field width, the representation is padded with fill characters. Fill characters are controlled by the `setfill` manipulator which takes one parameter `f` as the padding character.

For example, given:

```
ap_fixed<65, 32> aa = 123456;  
int precision = 5;  
cout<<setprecision(precision)<<setw(13)<<setfill('T')<<a<<endl;
```

The output is:

```
TTT1.2346e+05
```

Expressions Involving ap_[u]fixed<> types

Arbitrary precision fixed-point values can participate in expressions that use any operators supported by C/C++. After an arbitrary precision fixed-point type or variable is defined, their usage is the same as for any floating point type or variable in the C/C++ languages.

Observe the following caveats:

- Zero and Sign Extensions

All values of smaller bit-width are zero or sign-extended depending on the sign of the source value. You may need to insert casts to obtain alternative signs when assigning smaller bit-widths to larger.

- Truncations

Truncation occurs when you assign an arbitrary precision fixed-point of larger bit-width than the destination variable.

Class Methods, Operators, and Data Members

In general, any valid operation that can be done on a native C/C++ integer data type is supported (using operator overloading) for `ap_[u]fixed` types. In addition to these overloaded operators, some class specific operators and methods are included to ease bit-level operations.

Binary Arithmetic Operators

Addition

```
ap_[u]fixed::RType ap_[u]fixed::operator + (ap_[u]fixed op)
```

Adds an arbitrary precision fixed-point with a given operand `op`.

The operands can be any of the following integer types:

- `ap_[u]fixed`

- ap_[u]int
- C/C++

The result type `ap_[u]fixed::RType` depends on the type information of the two operands.

```
ap_fixed<76, 63> Result;
ap_fixed<5, 2> Val1 = 1.125;
ap_fixed<75, 62> Val2 = 6721.35595703125;

Result = Val1 + Val2; //Yields 6722.480957
```

Because `Val2` has the larger bit-width on both integer part and fraction part, the result type has the same bit-width and plus one to be able to store all possible result values.

Specifying the data's width controls resources by using the power functions, as shown below. In similar cases, Xilinx recommends specifying the width of the stored result instead of specifying the width of fixed point operations.

```
ap_ufixed<16,6> x=5;
ap_ufixed<16,7>y=hls::rsqrt<16,6>(x+x);
```

Subtraction

```
ap_[u]fixed::RType ap_[u]fixed::operator - (ap_[u]fixed op)
```

Subtracts an arbitrary precision fixed-point with a given operand `op`.

The result type `ap_[u]fixed::RType` depends on the type information of the two operands.

```
ap_fixed<76, 63> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val2 - Val1; // Yields 6720.23057
```

Because `Val2` has the larger bit-width on both integer part and fraction part, the result type has the same bit-width and plus one to be able to store all possible result values.

Multiplication

```
ap_[u]fixed::RType ap_[u]fixed::operator * (ap_[u]fixed op)
```

Multiplies an arbitrary precision fixed-point with a given operand `op`.

```
ap_fixed<80, 64> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;

Result = Val1 * Val2; // Yields 7561.525452
```

This shows the multiplication of Val1 and Val2. The result type is the sum of their integer part bit-width and their fraction part bit width.

Division

```
ap_[u]fixed::RType ap_[u]fixed::operator / (ap_[u]fixed op)
```

Divides an arbitrary precision fixed-point by a given operand op.

```
ap_fixed<84, 66> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;
Val2 / Val1; // Yields 5974.538628
```

This shows the division of Val2 and Val1. To preserve enough precision:

- The integer bit-width of the result type is sum of the integer bit-width of Val2 and the fraction bit-width of Val1.
- The fraction bit-width of the result type is equal to the fraction bit-width of Val2.

Bitwise Logical Operators

Bitwise OR

```
ap_[u]fixed::RType ap_[u]fixed::operator | (ap_[u]fixed op)
```

Applies a bitwise operation on an arbitrary precision fixed-point and a given operand op.

```
ap_fixed<75, 62> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;
Result = Val1 | Val2; // Yields 6271.480957
```

Bitwise AND

```
ap_[u]fixed::RType ap_[u]fixed::operator & (ap_[u]fixed op)
```

Applies a bitwise operation on an arbitrary precision fixed-point and a given operand op.

```
ap_fixed<75, 62> Result;
ap_fixed<5, 2> Val1 = 1625.153;
ap_fixed<75, 62> Val2 = 6721.355992351;
Result = Val1 & Val2; // Yields 1.00000
```

Bitwise XOR

```
ap_[u]fixed::RType ap_[u]fixed::operator ^ (ap_[u]fixed op)
```

Applies an `xor` bitwise operation on an arbitrary precision fixed-point and a given operand `op`.

```
ap_fixed<75, 62> Result;  
  
ap_fixed<5, 2> Val1 = 1625.153;  
ap_fixed<75, 62> Val2 = 6721.355992351;  
  
Result = Val1 ^ Val2; // Yields 6720.480957
```

Increment and Decrement Operators

Pre-Increment

```
ap_[u]fixed ap_[u]fixed::operator ++ ()
```

This operator function prefix increases an arbitrary precision fixed-point variable by 1.

```
ap_fixed<25, 8> Result;  
ap_fixed<8, 5> Val1 = 5.125;  
  
Result = ++Val1; // Yields 6.125000
```

Post-Increment

```
ap_[u]fixed ap_[u]fixed::operator ++ (int)
```

This operator function postfix:

- Increases an arbitrary precision fixed-point variable by 1.
- Returns the original val of this arbitrary precision fixed-point.

```
ap_fixed<25, 8> Result;  
ap_fixed<8, 5> Val1 = 5.125;  
  
Result = Val1++; // Yields 5.125000
```

Pre-Decrement

```
ap_[u]fixed ap_[u]fixed::operator -- ()
```

This operator function prefix decreases this arbitrary precision fixed-point variable by 1.

```
ap_fixed<25, 8> Result;  
ap_fixed<8, 5> Val1 = 5.125;  
  
Result = --Val1; // Yields 4.125000
```

Post-Decrement

```
ap_[u]fixed ap_[u]fixed::operator -- (int)
```

This operator function postfix:

- Decreases this arbitrary precision fixed-point variable by 1.
- Returns the original val of this arbitrary precision fixed-point.

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = Val1--; // Yields 5.125000
```

Unary Operators

Addition

```
ap_[u]fixed ap_[u]fixed::operator + ()
```

Returns a self copy of an arbitrary precision fixed-point variable.

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = +Val1; // Yields 5.125000
```

Subtraction

```
ap_[u]fixed::RType ap_[u]fixed::operator - ()
```

Returns a negative value of an arbitrary precision fixed-point variable.

```
ap_fixed<25, 8> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = -Val1; // Yields -5.125000
```

Equality Zero

```
bool ap_[u]fixed::operator ! ()
```

This operator function:

- Compares an arbitrary precision fixed-point variable with 0,
- Returns the result.

```
bool Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = !Val1; // Yields false
```

Bitwise Inverse

```
ap_[u]fixed::RType ap_[u]fixed::operator ~ ()
```

Returns a bitwise complement of an arbitrary precision fixed-point variable.

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val1 = 5.125;

Result = ~Val1; // Yields -5.25
```

Shift Operators

Unsigned Shift Left

```
ap_[u]fixed ap_[u]fixed::operator << (ap_uint<_W2> op)
```

This operator function:

- Shifts left by a given integer operand.
- Returns the result.

The operand can be a C/C++ integer type:

- char
- short
- int
- long

The return type of the shift left operation is the same width as the type being shifted.

Note: Shift does not support overflow or quantization modes.

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val << sh; // Yields -10.5
```

The bit-width of the result is ($W = 25, I = 15$). Because the shift left operation result type is same as the type of Val:

- The high order two bits of Val are shifted out.
- The result is -10.5.

If a result of 21.5 is required, Val must be cast to `ap_fixed<10, 7>` first -- for example, `ap_ufixed<10, 7>(Val)`.

Signed Shift Left

```
ap_[u]fixed ap_[u]fixed::operator << (ap_int<_W2> op)
```

This operator:

- Shifts left by a given integer operand.
- Returns the result.

The shift direction depends on whether the operand is positive or negative.

- If the operand is positive, a shift right is performed.
- If the operand is negative, a shift left (opposite direction) is performed.

The operand can be a C/C++ integer type:

- char
- short
- int
- long

The return type of the shift right operation is the same width as the type being shifted.

```
ap_fixed<25, 15, false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val << sh; // Shift left, yields -10.25

Sh = -2;
Result = Val << sh; // Shift right, yields 1.25
```

Unsigned Shift Right

```
ap_[u]fixed ap_[u]fixed::operator >> (ap_uint<_W2> op)
```

This operator function:

- Shifts right by a given integer operand.
- Returns the result.

The operand can be a C/C++ integer type:

- char
- short
- int
- long

The return type of the shift right operation is the same width as the type being shifted.

```
ap_fixed<25, 15> Result;
ap_fixed<8, 5> Val = 5.375;

ap_uint<4> sh = 2;

Result = Val >> sh; // Yields 1.25
```

If it is necessary to preserve all significant bits, extend fraction part bit-width of the `Val` first, for example `ap_fixed<10, 5>(Val)`.

Signed Shift Right

```
ap_[u]fixed ap_[u]fixed::operator >> (ap_int<_W2> op)
```

This operator:

- Shifts right by a given integer operand.
- Returns the result.

The shift direction depends on whether operand is positive or negative.

- If the operand is positive, a shift right performed.
- If operand is negative, a shift left (opposite direction) is performed.

The operand can be a C/C++ integer type (`char`, `short`, `int`, or `long`).

The return type of the shift right operation is the same width as type being shifted. For example:

```
ap_fixed<25, 15, false> Result;
ap_uint<8, 5> Val = 5.375;

ap_int<4> Sh = 2;
Result = Val >> sh; // Shift right, yields 1.25

Sh = -2;
Result = Val >> sh; // Shift left, yields -10.5

1.25
```

Relational Operators

Equality

```
bool ap_[u]fixed::operator == (ap_[u]fixed op)
```

This operator compares the arbitrary precision fixed-point variable with a given operand.

Returns `true` if they are equal and `false` if they are *not* equal.

The type of operand `op` can be `ap_[u]fixed`, `ap_int` or C/C++ integer types. For example:

```
bool Result;  
  
ap_ufixed<8, 5> Val1 = 1.25;  
ap_fixed<9, 4> Val2 = 17.25;  
ap_fixed<10, 5> Val3 = 3.25;  
  
Result = Val1 == Val2; // Yields true  
Result = Val1 == Val3; // Yields false
```

Inequality

```
bool ap_[u]fixed::operator != (ap_[u]fixed op)
```

This operator compares this arbitrary precision fixed-point variable with a given operand.

Returns `true` if they are *not* equal and `false` if they are equal.

The type of operand `op` can be:

- `ap_[u]fixed`
- `ap_int`
- C or C++ integer types

For example:

```
bool Result;  
  
ap_ufixed<8, 5> Val1 = 1.25;  
ap_fixed<9, 4> Val2 = 17.25;  
ap_fixed<10, 5> Val3 = 3.25;  
  
Result = Val1 != Val2; // Yields false  
Result = Val1 != Val3; // Yields true
```

Greater than or equal to

```
bool ap_[u]fixed::operator >= (ap_[u]fixed op)
```

This operator compares a variable with a given operand.

Returns `true` if they are equal or if the variable is greater than the operator and `false` otherwise.

The type of operand `op` can be `ap_[u]fixed`, `ap_int` or C/C++ integer types.

For example:

```
bool Result;  
  
ap_ufixed<8, 5> Val1 = 1.25;  
ap_fixed<9, 4> Val2 = 17.25;  
ap_fixed<10, 5> Val3 = 3.25;  
  
Result = Val1 >= Val2; // Yields true  
Result = Val1 >= Val3; // Yields false
```

Less than or equal to

```
bool ap_[u]fixed::operator <= (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is equal to or less than the operand and `false` if not.

The type of operand `op` can be `ap_[u]fixed`, `ap_int` or C/C++ integer types.

For example:

```
bool Result;  
  
ap_ufixed<8, 5> Val1 = 1.25;  
ap_fixed<9, 4> Val2 = 17.25;  
ap_fixed<10, 5> Val3 = 3.25;  
  
Result = Val1 <= Val2; // Yields true  
Result = Val1 <= Val3; // Yields true
```

Greater than

```
bool ap_[u]fixed::operator > (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is greater than the operand and `false` if not.

The type of operand `op` can be `ap_[u]fixed`, `ap_int`, or C/C++ integer types.

For example:

```
bool Result;  
  
ap_ufixed<8, 5> Val1 = 1.25;  
ap_fixed<9, 4> Val2 = 17.25;  
ap_fixed<10, 5> Val3 = 3.25;  
  
Result = Val1 > Val2; // Yields false  
Result = Val1 > Val3; // Yields false
```

Less than

```
bool ap_[u]fixed::operator < (ap_[u]fixed op)
```

This operator compares a variable with a given operand, and return `true` if it is less than the operand and `false` if not.

The type of operand `op` can be `ap_[u]fixed`, `ap_int`, or C/C++ integer types. For example:

```
bool Result;
ap_ufixed<8, 5> Val1 = 1.25;
ap_fixed<9, 4> Val2 = 17.25;
ap_fixed<10, 5> Val3 = 3.25;

Result = Val1 < Val2; // Yields false
Result = Val1 < Val3; // Yields true
```

Bit Operator

Bit-Select and Set

```
af_bit_ref ap_[u]fixed::operator [] (int bit)
```

This operator selects one bit from an arbitrary precision fixed-point value and returns it.

The returned value is a reference value that can set or clear the corresponding bit in the `ap_[u]fixed` variable. The `bit` argument must be an integer value and it specifies the index of the bit to select. The least significant bit has index 0. The highest permissible index is one less than the bit-width of this `ap_[u]fixed` variable.

The result type is `af_bit_ref` with a value of either 0 or 1. For example:

```
ap_int<8, 5> Value = 1.375;

Value[3]; // Yields 1
Value[4]; // Yields 0

Value[2] = 1; // Yields 1.875
Value[3] = 0; // Yields 0.875
```

Bit Range

```
af_range_ref af_(u)fixed::range (unsigned Hi, unsigned Lo)
af_range_ref af_(u)fixed::operator [] (unsigned Hi, unsigned Lo)
```

This operation is similar to bit-select operator `[]` except that it operates on a range of bits instead of a single bit.

It selects a group of bits from the arbitrary precision fixed-point variable. The `Hi` argument provides the upper range of bits to be selected. The `Lo` argument provides the lowest bit to be selected. If `Lo` is larger than `Hi` the bits selected are returned in the reverse order.

The return type `af_range_ref` represents a reference in the range of the `ap_[u]fixed` variable specified by `Hi` and `Lo`. For example:

```
ap_uint<4> Result = 0;
ap_ufixed<4, 2> Value = 1.25;
ap_uint<8> Repl = 0xAA;

Result = Value.range(3, 0); // Yields: 0x5
Value(3, 0) = Repl(3, 0); // Yields: -1.5

// when Lo > Hi, return the reverse bits string
Result = Value.range(0, 3); // Yields: 0xA
```

Range Select

```
af_range_ref af_(u)fixed::range()
af_range_ref af_(u)fixed::operator []
```

This operation is the special case of the range select operator `[]`. It selects all bits from this arbitrary precision fixed-point value in the normal order.

The return type `af_range_ref` represents a reference to the range specified by `Hi = W - 1` and `Lo = 0`. For example:

```
ap_uint<4> Result = 0;

ap_ufixed<4, 2> Value = 1.25;
ap_uint<8> Repl = 0xAA;

Result = Value.range(); // Yields: 0x5
Value() = Repl(3, 0); // Yields: -1.5
```

Length

```
int ap_[u]fixed::length()
```

This function returns an integer value that provides the number of bits in an arbitrary precision fixed-point value. It can be used with a type or a value. For example:

```
ap_ufixed<128, 64> My128APFixed;

int bitwidth = My128APFixed.length(); // Yields 128
```

Explicit Conversion Methods

Fixed to Double

```
double ap_[u]fixed::to_double()
```

This member function returns this fixed-point value in form of IEEE double precision format. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
double Result;  
  
Result = MyAPFixed.to_double(); // Yields 333.789
```

Fixed to Float

```
float ap_[u]fixed::to_float()
```

This member function returns this fixed-point value in form of IEEE float precision format. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
float Result;  
  
Result = MyAPFixed.to_float(); // Yields 333.789
```

Fixed to Half-Precision Floating Point

```
half ap_[u]fixed::to_half()
```

This member function return this fixed-point value in form of HLS half-precision (16-bit) float precision format. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
half Result;  
  
Result = MyAPFixed.to_half(); // Yields 333.789
```

Fixed to ap_int

```
ap_int ap_[u]fixed::to_ap_int ()
```

This member function explicitly converts this fixed-point value to `ap_int` that captures all integer bits (fraction bits are truncated). For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;  
ap_uint<77> Result;  
  
Result = MyAPFixed.to_ap_int(); //Yields 333
```

Fixed to Integer

```
int ap_[u]fixed::to_int ()  
unsigned ap_[u]fixed::to_uint ()  
ap_slong ap_[u]fixed::to_int64 ()  
ap_ulong ap_[u]fixed::to_uint64 ()
```

This member function explicitly converts this fixed-point value to C built-in integer types. For example:

```
ap_ufixed<256, 77> MyAPFixed = 333.789;
unsigned int Result;

Result = MyAPFixed.to_uint(); //Yields 333

unsigned long long Result;
Result = MyAPFixed.to_uint64(); //Yields 333
```



RECOMMENDED: Xilinx recommends that you explicitly call member functions instead of using C-style cast to convert `ap_[u]fixed` to other data types.

Compile Time Access to Data Type Attributes

The `ap_[u]fixed<>` types are provided with several static members that allow the size and configuration of data types to be determined at compile time. The data type is provided with the static const members: `width`, `iwidth`, `qmode` and `omode`:

```
static const int width = _AP_W;
static const int iwidth = _AP_I;
static const ap_q_mode qmode = _AP_Q;
static const ap_o_mode omode = _AP_O;
```

You can use these data members to extract the following information from any existing `ap_[u]fixed<>` data type:

- `width`: The width of the data type.
- `iwidth`: The width of the integer part of the data type.
- `qmode`: The quantization mode of the data type.
- `omode`: The overflow mode of the data type.

For example, you can use these data members to extract the data width of an existing `ap_[u]fixed<>` data type to create another `ap_[u]fixed<>` data type at compile time.

The following example shows how the size of variable `Res` is automatically defined as 1-bit greater than variables `Val1` and `Val2` with the same quantization modes:

```
// Definition of basic data type
#define INPUT_DATA_WIDTH 12
#define IN_INTG_WIDTH 6
#define IN_QMODE AP_RND_ZERO
#define IN_OMODE AP_WRAP
typedef ap_fixed<INPUT_DATA_WIDTH, IN_INTG_WIDTH, IN_QMODE, IN_OMODE>
data_t;
// Definition of variables
data_t Val1, Val2;
// Res is automatically sized at run-time to be 1-bit greater than
```

```
INPUT_DATA_WIDTH
// The bit growth in Res will be in the integer bits
ap_int<data_t::width+1, data_t::iwidth+1, data_t::qmode, data_t::omode> Res
= Val1 +
Val2;
```

This ensures that Vitis HLS correctly models the bit-growth caused by the addition even if you update the value of INPUT_DATA_WIDTH, IN_INTG_WIDTH, or the quantization modes for data_t.

Vitis HLS Math Library

The Vitis™ HLS Math Library (`hls_math.h`) provides coverage of math functions from C++ (`cmath`) libraries, and can be used in both C simulation and synthesis. It offers floating-point (single-precision, double-precision, and half-precision) for all functions and fixed-point support for the majority of the functions. The functions in `hls_math.h` is grouped in `hls` namespace, and can be used as in-place replacement of function of `std` namespace from the standard C++ math library (`cmath`).



IMPORTANT! Using `hls_math.h` header in C code is not supported.

HLS Math Library Accuracy

The HLS math functions are implemented as synthesizable bit-approximate functions from the `hls_math.h` library. Bit-approximate HLS math library functions do not provide the same accuracy as the standard C function. To achieve the desired result, the bit-approximate implementation might use a different underlying algorithm than the standard C math library version. The accuracy of the function is specified in terms of ULP (Unit of Least Precision). This difference in accuracy has implications for both C simulation and C/RTL co-simulation.

The ULP difference is typically in the range of 1-4 ULP.

- If the standard C math library is used in the C source code, there may be a difference between the C simulation and the C/RTL co-simulation due to the fact that some functions exhibit a ULP difference from the standard C math library.
- If the HLS math library is used in the C source code, there will be no difference between the C simulation and the C/RTL co-simulation. A C simulation using the HLS math library, may however differ from a C simulation using the standard C math library.

In addition, the following seven functions might show some differences, depending on the C standard used to compile and run the C simulation:

- `copysign`
- `fpclassify`
- `isinf`

- `isfinite`
- `isnan`
- `isnormal`
- `signbit`

C90 mode

Only `isinf`, `isnan`, and `copysign` are usually provided by the system header files, and they operate on doubles. In particular, `copysign` always returns a double result. This might result in unexpected results after synthesis if it must be returned to a float, because a double-to-float conversion block is introduced into the hardware.

C99 mode (-std=c99)

All seven functions are usually provided under the expectation that the system header files will redirect them to `__isnan(double)` and `__isnan(float)`. The usual GCC header files do not redirect `isnormal`, but implement it in terms of `fpclassify`.

C++ Using math.h

All seven are provided by the system header files, and they operate on doubles.

`copysign` always returns a double result. This might cause unexpected results after synthesis if it must be returned to a float, because a double-to-float conversion block is introduced into the hardware.

C++ Using cmath

Similar to C99 mode (`-std=c99`), except that:

- The system header files are usually different.
- The functions are properly overloaded for:
 - `float().snan(double)`
 - `isinf(double)`

`copysign` and `copysignf` are handled as built-ins even when using `namespace std;`.

C++ Using cmath and namespace std

No issues. Xilinx recommends using the following for best results:

- `-std=c99` for C
- `-fno-builtins` for C and C++

Note: To specify the C compile options, such as `-std=c99`, use the Tcl command `add_files` with the `-cflags` option. Alternatively, use the **Edit CFLAGS** button in the Project Settings dialog box.

HLS Math Library

The following functions are provided in the HLS math library. Each function supports half-precision (type `half`), single-precision (type `float`) and double precision (type `double`).



IMPORTANT! For each function `func` listed below, there is also an associated half-precision only function named `half_func` and single-precision only function named `funcf` provided in the library.

When mixing half-precision, single-precision and double-precision data types, check for common synthesis errors to prevent introducing type-conversion hardware in the final FPGA implementation.

Trigonometric Functions

acos	acospi	asin	asinpi
atan	atan2	atan2pi	cos
cospi	sin	sincos	sinpi
tan	tanpi		

Hyperbolic Functions

acosh	asinh	atanh	cosh
sinh	tanh		

Exponential Functions

exp	exp10	exp2	expm1
frexp	ldexp	modf	

Logarithmic Functions

ilogb	log	log10	log1p

Power Functions

cbrt	hypot	pow	rsqrt
sqrt			

Error Functions

erf	erfc

Rounding Functions

ceil	floor	llrint	llround
lrint	lround	nearbyint	rint
round	trunc		

Remainder Functions

fmod	remainder	remquo
------	-----------	--------

Floating-point

copysign	nan	nextafter	nexttoward
----------	-----	-----------	------------

Difference Functions

fdim	fmax	fmin	maxmag
minmag			

Other Functions

abs	divide	fabs	fma
fract	mad	recip	

Classification Functions

fpclassify	isfinite	isinf	isnan
isnormal	signbit		

Comparison Functions

isgreater	isgreaterequal	isless	islessequal
islessgreater	isunordered		

Relational Functions

all	any	bitselect	isequal
isnotequal	isordered	select	

Fixed-Point Math Functions

Fixed-point implementations are also provided for the following math functions.

All fixed-point math functions support `ap_[u]fixed` and `ap_[u]int` data types with following bit-width specification,

1. `ap_fixed<W, I>` where $I \leq 33$ and $W-I \leq 32$
2. `ap_ufixed<W, I>` where $I \leq 32$ and $W-I \leq 32$
3. `ap_int<I>` where $I \leq 33$
4. `ap_uint<I>` where $I \leq 32$

Trigonometric Functions

<code>cos</code>	<code>sin</code>	<code>tan</code>	<code>acos</code>	<code>asin</code>	<code>atan</code>	<code>atan2</code>	<code>sincos</code>
<code>cospf</code>	<code>sinpf</code>						

Hyperbolic Functions

<code>cosh</code>	<code>sinh</code>	<code>tanh</code>	<code>acosh</code>	<code>asinh</code>	<code>atanh</code>
-------------------	-------------------	-------------------	--------------------	--------------------	--------------------

Exponential Functions

<code>exp</code>	<code>frexp</code>	<code>modf</code>	<code>exp2</code>	<code>expm1</code>
------------------	--------------------	-------------------	-------------------	--------------------

Logarithmic Functions

<code>log</code>	<code>log10</code>	<code>ilogb</code>	<code>log1p</code>
------------------	--------------------	--------------------	--------------------

Power Functions

<code>pow</code>	<code>sqrt</code>	<code>rsqrt</code>	<code>cbrt</code>	<code>hypot</code>
------------------	-------------------	--------------------	-------------------	--------------------

Error Functions

<code>erf</code>	<code>erfc</code>
------------------	-------------------

Rounding Functions

<code>ceil</code>	<code>floor</code>	<code>trunc</code>	<code>round</code>	<code>rint</code>	<code>nearbyint</code>
-------------------	--------------------	--------------------	--------------------	-------------------	------------------------

Floating Point

<code>nextafter</code>	<code>nexttoward</code>
------------------------	-------------------------

Difference Functions

<code>erf</code>	<code>erfc</code>	<code>fdim</code>	<code>fmax</code>	<code>fmin</code>	<code>maxmag</code>	<code>minmag</code>
------------------	-------------------	-------------------	-------------------	-------------------	---------------------	---------------------

Other Functions

fabs

recip

abs

fract

divide

Classification Functions

signbit

Comparison Functions

isgreater

isgreaterequal

isless

islessequal

islessgreater

Relational Functions

isequal

isnotequal

any

all

bitselect

The fixed-point type provides a slightly-less accurate version of the function value, but a smaller and faster RTL implementation.

The methodology for implementing a math function with a fixed-point data types is:

1. Determine if a fixed-point implementation is supported.
2. Update the math functions to use `ap_fixed` types.
3. Perform C simulation to validate the design still operates with the required precision. The C simulation is performed using the same bit-accurate types as the RTL implementation.
4. Synthesize the design.

For example, a fixed-point implementation of the function `sin` is specified by using fixed-point types with the math function as follows:

```
#include "hls_math.h"
#include "ap_fixed.h"

ap_fixed<32,2> my_input, my_output;

my_input = 24.675;
my_output = sin(my_input);
```

When using fixed-point math functions, the result type must have the same width and integer bits as the input.

Verification and Math Functions

If the standard C math library is used in the C source code, the C simulation results and the C/RTL co-simulation results may be different: if any of the math functions in the source code have an ULP difference from the standard C math library it may result in differences when the RTL is simulated.

If the `hls_math.h` library is used in the C source code, the C simulation and C/RTL co-simulation results are identical. However, the results of C simulation using `hls_math.h` are not the same as those using the standard C libraries. The `hls_math.h` library simply ensures the C simulation matches the C/RTL co-simulation results. In both cases, the same RTL implementation is created. The following explains each of the possible options which are used to perform verification when using math functions.

Verification Option 1: Standard Math Library and Verify Differences

In this option, the standard C math libraries are used in the source code. If any of the functions synthesized do have exact accuracy the C/RTL co-simulation is different than the C simulation. The following example highlights this approach.

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

typedef float data_t;

data_t cpp_math(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}
```

In this case, the results between C simulation and C/RTL co-simulation are different. Keep in mind when comparing the outputs of simulation, any results written from the test bench are written to the working directory where the simulation executes:

- C simulation: Folder `<project>/<solution>/csim/build`
- C/RTL co-simulation: Folder `<project>/<solution>/sim/<RTL>`

where `<project>` is the project folder, `<solution>` is the name of the solution folder and `<RTL>` is the type of RTL verified (Verilog or VHDL). The following figure shows a typical comparison of the pre-synthesis results file on the left-hand side and the post-synthesis RTL results file on the right-hand side. The output is shown in the third column.

Figure 128: Pre-Synthesis and Post-Synthesis Simulation Differences

	result.dat	proj_cpp_math.prj/solution1/sim/systemc/result.dat
1	0.000000000000000	0.00999999776483
2	1.000000000000000	0.10999999403954
3	2.000000000000000	0.209999993443489
4	3.000000000000000	0.310000002384186
5	4.000000000000000	0.409999995423721
6	5.000000000000000	0.509999990463257
7	6.000000000000000	0.61000014305115
8	7.000000000000000	0.71000038146973
9	8.000000000000000	0.81000061988831
10	9.000000000000000	0.91000085830688
11	10.0000000000000	1.01000109672546
12	11.0000000000000	1.1000033514404
13	12.0000000000000	1.21000157356262
14	13.0000000000000	1.31000181198120
15	14.0000000000000	1.410000205039978
16	15.0000000000000	1.510000228881836
17	16.0000000000000	1.610000252723694
18	17.0000000000000	1.710000276565552
19	18.0000000000000	1.810000300407410
20	19.0000000000000	1.910000324249268
21	20.0000000000000	2.010000228881836
22	21.0000000000000	2.110000133514404
23	22.0000000000000	2.210000038146973
24	23.0000000000000	2.30999942779541
25	24.0000000000000	2.409999847412109
26	25.0000000000000	2.509999752044678
27	26.0000000000000	2.609999656677246
28	27.0000000000000	2.709999561309814
29	28.0000000000000	2.80999465942383
30	29.0000000000000	2.90999370574951

The results of pre-synthesis simulation and post-synthesis simulation differ by fractional amounts. You must decide whether these fractional amounts are acceptable in the final RTL implementation.

The recommended flow for handling these differences is using a test bench that checks the results to ensure that they lie within an acceptable error range. This can be accomplished by creating two versions of the same function, one for synthesis and one as a reference version. In this example, only function `cpp_math` is synthesized.

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

typedef float data_t;

data_t cpp_math(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}

data_t cpp_math_sw(data_t angle) {
    data_t s = sinf(angle);
    data_t c = cosf(angle);
    return sqrtf(s*s+c*c);
}
```

The test bench to verify the design compares the outputs of both functions to determine the difference, using variable `diff` in the following example. During C simulation both functions produce identical outputs. During C/RTL co-simulation function `cpp_math` produces different results and the difference in results are checked.

```
int main() {
    data_t angle = 0.01;
    data_t output, exp_output, diff;
    int retval=0;

    for (data_t i = 0; i <= 250; i++) {
        output = cpp_math(angle);
        exp_output = cpp_math_sw(angle);

        // Check for differences
        diff = ( (exp_output > output) ? exp_output - output : output - exp_output );
        if (diff > 0.0000005) {
            printf("Difference %.10f exceeds tolerance at angle %.10f \n", diff, angle);
            retval=1;
        }

        angle = angle + .1;
    }

    if (retval != 0) {
        printf("Test failed !!!\n");
        retval=1;
    } else {
        printf("Test passed !\n");
    }
    // Return 0 if the test passes
    return retval;
}
```

If the margin of difference is lowered to 0.00000005, this test bench highlights the margin of error during C/RTL co-simulation:

```
Difference 0.0000000596 at angle 1.1100001335
Difference 0.0000000596 at angle 1.2100001574
Difference 0.0000000596 at angle 1.5100002289
Difference 0.0000000596 at angle 1.6100002527
etc..
```

When using the standard C math libraries (`math.h` and `cmath.h`) create a “smart” test bench to verify any differences in accuracy are acceptable.

Verification Option 2: HLS Math Library and Validate Differences

An alternative verification option is to convert the source code to use the HLS math library. With this option, there are no differences between the C simulation and C/RTL co-simulation results. The following example shows how the code above is modified to use the `hls_math.h` library.

Note: This option is only available in C++.

- Include the `hls_math.h` header file.
- Replace the math functions with the equivalent `hls::` function.

```
#include <cmath>
#include "hls_math.h"
#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

typedef float data_t;

data_t cpp_math(data_t angle) {
    data_t s = hls::sinf(angle);
    data_t c = hls::cosf(angle);
    return hls::sqrtf(s*s+c*c);
}
```

Verification Option 3: HLS Math Library File and Validate Differences

Including the HLS math library file `lib_hlsm.cpp` as a design file ensures Vitis HLS uses the HLS math library for C simulation. This option is identical to option2 however it does not require the C code to be modified.

The HLS math library file is located in the `src` directory in the Vitis HLS installation area. Simply copy the file to your local folder and add the file as a standard design file.

Note: This option is only available in C++.

As with option 2, with this option there is now a difference between the C simulation results using the HLS math library file and those previously obtained without adding this file. These difference should be validated with C simulation using a “smart” test bench similar to option 1.

Common Synthesis Errors

The following are common use errors when synthesizing math functions. These are often (but not exclusively) caused by converting C functions to C++ to take advantage of synthesis for math functions.

C++ cmath.h

If the C++ `cmath.h` header file is used, the floating point functions (for example, `sinf` and `cosf`) can be used. These result in 32-bit operations in hardware. The `cmath.h` header file also overloads the standard functions (for example, `sin` and `cos`) so they can be used for float and double types.

C math.h

If the `C math.h` library is used, the single-precision functions (for example, `sinf` and `cosf`) are required to synthesize 32-bit floating point operations. All standard function calls (for example, `sin` and `cos`) result in doubles and 64-bit double-precision operations being synthesized.

Cautions

When converting C functions to C++ to take advantage of `math.h` support, be sure that the new C++ code compiles correctly before synthesizing with Vitis HLS. For example, if `sqrtf()` is used in the code with `math.h`, it requires the following code `extern` added to the C++ code to support it:

```
#include <math.h>
extern "C" float sqrtf(float);
```

To avoid unnecessary hardware caused by type conversion, follow the warnings on mixing double and float types discussed in [Floats and Doubles](#).

HLS Stream Library

Streaming data is a type of data transfer in which data samples are sent in sequential order starting from the first sample. Streaming requires no address management.

Modeling designs that use streaming data can be difficult in C. The approach of using pointers to perform multiple read and/or write accesses can introduce issues, because there are implications for the type qualifier and how the test bench is constructed.

Vitis HLS provides a C++ template class `hls::stream<>` for modeling streaming data structures. The streams implemented with the `hls::stream<>` class have the following attributes.

- In the C code, an `hls::stream<>` behaves like a FIFO of infinite depth. There is no requirement to define the size of an `hls::stream<>`.
- They are read from and written to sequentially. That is, after data is read from an `hls::stream<>`, it cannot be read again.
- An `hls::stream<>` on the top-level interface is by default implemented with an `ap_fifo` interface.
- There are two possible stream declarations:
 - `hls::stream<Type>`: specify the data type for the stream.

An `hls::stream<>` internal to the design is implemented as a FIFO with a default depth of 2. The **STREAM** pragma or directive can be used to change the depth.

- `hls::stream<Type, Depth>`: specify the data type for the stream, and the FIFO depth.

Set the depth to prevent stalls. If any task in the design can produce or consume samples at a greater rate than the specified depth, the FIFOs might become empty (or full) resulting in stalls, because it is unable to read (or write).

This section shows how the `hls::stream<>` class can more easily model designs with streaming data. The topics in this section provide:

- An overview of modeling with streams and the RTL implementation of streams.
- Rules for global stream variables.
- How to use streams.
- Blocking reads and writes.

- Non-Blocking Reads and writes.
- Controlling the FIFO depth.

Note: The `hls::stream` class should always be passed between functions as a C++ reference argument. For example, `&my_stream`.



IMPORTANT! The `hls::stream` class is only used in C++ designs. Array of streams is not supported.

C Modeling and RTL Implementation

Streams are modeled as an infinite queue in software (and in the test bench during RTL co-simulation). There is no need to specify any depth to simulate streams in C++. Streams can be used inside functions and on the interface to functions. Internal streams may be passed as function parameters.

Streams can be used only in C++ based designs. Each `hls::stream<>` object must be written by a single process and read by a single process.

If an `hls::stream` is used on the top-level interface, it is by default implemented in the RTL as a FIFO interface (`ap_fifo`) but may be optionally implemented as a handshake interface (`ap_hs`) or an AXI4-Stream interface (`axis`).

If an `hls::stream` is used inside the design function and synthesized into hardware, it is implemented as a FIFO with a default depth of 2. In some cases, such as when interpolation is used, the depth of the FIFO might have to be increased to ensure the FIFO can hold all the elements produced by the hardware. Failure to ensure the FIFO is large enough to hold all the data samples generated by the hardware can result in a stall in the design (seen in C/RTL co-simulation and in the hardware implementation). The depth of the FIFO can be adjusted using the STREAM directive with the `depth` option. An example of this is provided in the example design `hls_stream`.



IMPORTANT! Ensure `hls::stream` variables are correctly sized when used in the default non-DATAFLOW regions.

If an `hls::stream` is used to transfer data between tasks (sub-functions or loops), you should immediately consider implementing the tasks in a DATAFLOW region where data streams from one task to the next. The default (non-DATAFLOW) behavior is to complete each task before starting the next task, in which case the FIFOs used to implement the `hls::stream` variables must be sized to ensure they are large enough to hold all the data samples generated by the producer task. Failure to increase the size of the `hls::stream` variables results in the error below:

```
ERROR: [XFORM 203-733] An internal stream xxxx.xxxx.V.user.V' with default
size is
used in a non-dataflow region, which may result in deadlock. Please
consider to
resize the stream using the directive 'set_directive_stream' or the 'HLS
stream'
pragma.
```

This error informs you that in a non-DATAFLOW region, the default FIFOs depth of 2 may not be large enough to hold all the data samples written to the FIFO by the producer task, and deadlock may occur.

Global and Local Streams

Streams may be defined either locally or globally. Local streams are always implemented as internal FIFOs. Global streams can be implemented as internal FIFOs or ports:

- Globally-defined streams that are only read from, or only written to, are inferred as external ports of the top-level RTL block.
- Globally-defined streams that are both read from and written to (in the hierarchy below the top-level function) are implemented as internal FIFOs.

Streams defined in the global scope follow the same rules as any other global variables.

Using HLS Streams

To use `hls::stream<>` objects, include the header file `hls_stream.h`. Streaming data objects are defined by specifying the type and variable name. In this example, a 128-bit unsigned integer type is defined and used to create a stream variable called `my_wide_stream`.

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

typedef ap_uint<128> uint128_t; // 128-bit user defined type
hls::stream<uint128_t> my_wide_stream; // A stream declaration
```

Streams must use scoped naming. Xilinx recommends using the scoped `hls::` naming shown in the example above. However, if you want to use the `hls` namespace, you can rewrite the preceding example as:

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;

typedef ap_uint<128> uint128_t; // 128-bit user defined type
stream<uint128_t> my_wide_stream; // hls:: no longer required
```

Given a stream specified as `hls::stream<T>`, the type `T` can be:

- Any C++ native data type
- A Vitis HLS arbitrary precision type (for example, `ap_int<>`, `ap_ufixed<>`)
- A user-defined struct containing either of the above types

Note: General user-defined classes (or structures) that contain methods (member functions) should not be used as the type (`T`) for a stream variable.

A stream can also be specified as `hls::stream<Type, Depth>`, where `Depth` indicates the depth of the FIFO needed in the verification adapter that the HLS tool creates for RTL co-simulation.

Streams can be optionally named. Providing a name for the stream allows the name to be used in reporting. For example, Vitis HLS automatically checks to ensure all elements from an input stream are read during simulation. Given the following two streams:

```
stream<uint8_t> bytestr_in1;
stream<uint8_t> bytestr_in2("input_stream2");
```

Any warning on elements of the streams are reported as follows, where it is clear that `input_stream2` refers to `bytestr_in2`:

```
WARNING: Hls::stream 'hls::stream<unsigned char>.1' contains leftover data,
which
may result in RTL simulation hanging.
WARNING: Hls::stream 'input_stream2' contains leftover data, which may
result in RTL
simulation hanging.
```

When streams are passed into and out of functions, they must be passed-by-reference as in the following example:

```
void stream_function (
    hls::stream<uint8_t> &strm_out,
    hls::stream<uint8_t> &strm_in,
    uint16_t strm_len
)
```

Vitis HLS supports both blocking and non-blocking access methods.

A complete design example using streams is provided in the Vitis HLS examples. Refer to the `hls_stream` example in the design examples available from the Vitis IDE welcome screen.

Blocking Reads and Writes

The basic accesses to an `hls::stream<>` object are blocking reads and writes. These are accomplished using class methods. These methods stall (block) execution if a read is attempted on an empty stream FIFO, a write is attempted to a full stream FIFO, or until a full handshake is accomplished for a stream mapped to an `ap_hs` interface protocol.

A stall can be observed in C/RTL co-simulation as the continued execution of the simulator without any progress in the transactions. The following shows a classic example of a stall situation, where the RTL simulation time keeps increasing, but there is no progress in the inter or intra transactions:

```
// RTL Simulation : "Inter-Transaction Progress" ["Intra-Transaction
Progress"] @
"Simulation Time"
////////////////////////////// //////////////////////////////// //////////////////////////////// ////////////////////////////////
/////
// RTL Simulation : 0 / 1 [0.00%] @ "110000"
// RTL Simulation : 0 / 1 [0.00%] @ "202000"
// RTL Simulation : 0 / 1 [0.00%] @ "404000"
```

Blocking Write Methods

In this example, the value of variable `src_var` is pushed into the stream.

```
// Usage of void write(const T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

my_stream.write(src_var);
```

The `<<` operator is overloaded such that it may be used in a similar fashion to the stream insertion operators for C++ stream (for example, iostreams and filestreams). The `hls::stream<>` object to be written to is supplied as the left-hand side argument and the value to be written as the right-hand side.

```
// Usage of void operator << (T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

my_stream << src_var;
```

Blocking Read Methods

This method reads from the head of the stream and assigns the values to the variable `dst_var`.

```
// Usage of void read(T &rdata)

hls::stream<int> my_stream;
int dst_var;

my_stream.read(dst_var);
```

Alternatively, the next object in the stream can be read by assigning (using for example `=`, `+=`) the stream to an object on the left-hand side:

```
// Usage of T read(void)

hls::stream<int> my_stream;

int dst_var = my_stream.read();
```

The `>>` operator is overloaded to allow use similar to the stream extraction operator for C++ stream (for example, `iostreams` and `filestreams`). The `hls::stream` is supplied as the LHS argument and the destination variable the RHS.

```
// Usage of void operator >> (T & rdata)

hls::stream<int> my_stream;
int dst_var;

my_stream >> dst_var;
```

Non-Blocking Reads and Writes

Non-blocking write and read methods are also provided. These allow execution to continue even when a read is attempted on an empty stream or a write to a full stream.

These methods return a Boolean value indicating the status of the access (`true` if successful, `false` otherwise). Additional methods are included for testing the status of an `hls::stream<>` stream.



IMPORTANT! Non-blocking behavior is only supported on interfaces using the `ap_fifo` protocol. More specifically, the AXI-Stream standard and the Xilinx `ap_hs` IO protocol do not support non-blocking accesses.

During C simulation, streams have an infinite size. It is therefore not possible to validate with C simulation if the stream is full. These methods can be verified only during RTL simulation when the FIFO sizes are defined (either the default size of 1, or an arbitrary size defined with the `STREAM` directive).



IMPORTANT! If the design is specified to use the block-level I/O protocol `ap_ctrl_none` and the design contains any `hls::stream` variables that employ non-blocking behavior, C/RTL co-simulation is not guaranteed to complete.

Non-Blocking Writes

This method attempts to push variable `src_var` into the stream `my_stream`, returning a boolean `true` if successful. Otherwise, `false` is returned and the queue is unaffected.

```
// Usage of void write_nb(const T & wdata)

hls::stream<int> my_stream;
int src_var = 42;

if (my_stream.write_nb(src_var)) {
    // Perform standard operations
    ...
} else {
    // Write did not occur
    return;
}
```

Fullness Test

```
bool full(void)
```

Returns `true`, if and only if the `hls::stream<>` object is full.

```
// Usage of bool full(void)

hls::stream<int> my_stream;
int src_var = 42;
bool stream_full;

stream_full = my_stream.full();
```

Non-Blocking Read

```
bool read_nb(T & rdata)
```

This method attempts to read a value from the stream, returning `true` if successful. Otherwise, `false` is returned and the queue is unaffected.

```
// Usage of void read_nb(const T & wdata)

hls::stream<int> my_stream;
int dst_var;

if (my_stream.read_nb(dst_var)) {
    // Perform standard operations
```

```

...
} else {
    // Read did not occur
    return;
}

```

Emptiness Test

```
bool empty(void)
```

Returns true if the `hls::stream<>` is empty.

```

// Usage of bool empty(void)

hls::stream<int> my_stream;
int dst_var;
bool stream_empty;

stream_empty = my_stream.empty();

```

The following example shows how a combination of non-blocking accesses and full/empty tests can provide error handling functionality when the RTL FIFOs are full or empty:

```

#include "hls_stream.h"
using namespace hls;

typedef struct {
    short    data;
    bool     valid;
    bool     invert;
} input_interface;

bool invert(stream<input_interface>& in_data_1,
            stream<input_interface>& in_data_2,
            stream<short>& output
        ) {
    input_interface in;
    bool full_n;

    // Read an input value or return
    if (!in_data_1.read_nb(in))
        if (!in_data_2.read_nb(in))
            return false;

    // If the valid data is written, return not-full (full_n) as true
    if (in.valid) {
        if (in.invert)
            full_n = output.write_nb(~in.data);
        else
            full_n = output.write_nb(in.data);
    }
    return full_n;
}

```

Controlling the RTL FIFO Depth

For most designs using streaming data, the default RTL FIFO depth of 2 is sufficient. Streaming data is generally processed one sample at a time.

For multirate designs in which the implementation requires a FIFO with a depth greater than 2, you must determine (and set using the STREAM directive) the depth necessary for the RTL simulation to complete. If the FIFO depth is insufficient, RTL co-simulation stalls.

Because stream objects cannot be viewed in the GUI directives pane, the STREAM directive cannot be applied directly in that pane.

Right-click the function in which an `hls::stream<>` object is declared (or is used, or exists in the argument list) to:

- Select the STREAM directive.
- Populate the `variable` field manually with name of the stream variable.

Alternatively, you can:

- Specify the STREAM directive manually in the `directives.tcl` file, or
- Add it as a pragma in `source`.

C/RTL Co-Simulation Support

The Vitis HLS C/RTL co-simulation feature does not support structures or classes containing `hls::stream<>` members in the top-level interface. Vitis HLS supports these structures or classes for synthesis.

```
typedef struct {
    hls::stream<uint8_t> a;
    hls::stream<uint16_t> b;
} strm_struct_t;

void dut_top(strm_struct_t indata, strm_struct_t outdata) { }
```

These restrictions apply to both top-level function arguments and globally declared objects. If structs of streams are used for synthesis, the design must be verified using an external RTL simulator and user-created HDL test bench. There are no such restrictions on `hls::stream<>` objects with strictly internal linkage.

HLS IP Libraries

Vitis™ HLS provides C++ libraries to implement a number of Xilinx® IP blocks. The C libraries allow the following Xilinx IP blocks to be directly inferred from the C++ source code ensuring a high-quality implementation in the FPGA.

Table 42: HLS IP Libraries

Library Header File	Description
hls_fft.h	Allows the Xilinx LogiCORE IP FFT to be simulated in C and implemented using the Xilinx LogiCORE block.
hls_fir.h	Allows the Xilinx LogiCORE IP FIR to be simulated in C and implemented using the Xilinx LogiCORE block.
hls_dds.h	Allows the Xilinx LogiCORE IP DDS to be simulated in C and implemented using the Xilinx LogiCORE block.
ap_shift_reg.h	Provides a C++ class to implement a shift register which is implemented directly using a Xilinx SRL primitive.

FFT IP Library

The Xilinx FFT IP block can be called within a C++ design using the library `hls_fft.h`. This section explains how the FFT can be configured in your C++ code.



RECOMMENDED: Xilinx highly recommends that you review the *Fast Fourier Transform LogiCORE IP Product Guide (PG109)* for information on how to implement and use the features of the IP.

To use the FFT in your C++ code:

1. Include the `hls_fft.h` library in the code
2. Set the default parameters using the predefined struct `hls::ip_fft::params_t`
3. Define the runtime configuration
4. Call the FFT function
5. Optionally, check the runtime status

The following code examples provide a summary of how each of these steps is performed. Each step is discussed in more detail below.

First, include the FFT library in the source code. This header file resides in the include directory in the Vitis HLS installation area which is automatically searched when Vitis HLS executes.

```
#include "hls_fft.h"
```

Define the static parameters of the FFT. This includes such things as input width, number of channels, type of architecture, which do not change dynamically. The FFT library includes a parameterization struct `hls::ip_fft::params_t`, which can be used to initialize all static parameters with default values.

In this example, the default values for output ordering and the widths of the configuration and status ports are over-ridden using a user-defined struct `param1` based on the predefined struct.

```
struct param1 : hls::ip_fft::params_t {  
    static const unsigned ordering_opt = hls::ip_fft::natural_order;  
    static const unsigned config_width = FFT_CONFIG_WIDTH;  
    static const unsigned status_width = FFT_STATUS_WIDTH;  
};
```

Define types and variables for both the runtime configuration and runtime status. These values can be dynamic and are therefore defined as variables in the C code which can change and are accessed through APIs.

```
typedef hls::ip_fft::config_t<param1> config_t;  
typedef hls::ip_fft::status_t<param1> status_t;  
config_t fft_config1;  
status_t fft_status1;
```

Next, set the runtime configuration. This example sets the direction of the FFT (Forward or Inverse) based on the value of variable “direction” and also set the value of the scaling schedule.

```
fft_config1.setDir(direction);  
fft_config1.setSch(0x2AB);
```

Call the FFT function using the HLS namespace with the defined static configuration (`param1` in this example). The function parameters are, in order, input data, output data, output status and input configuration.

```
hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);
```

Finally, check the output status. This example checks the overflow flag and stores the results in variable “ovflo”.

```
*ovflo = fft_status1->getOvflo();
```

FFT Static Parameters

The static parameters of the FFT define how the FFT is configured and specifies the fixed parameters such as the size of the FFT, whether the size can be changed dynamically, whether the implementation is pipelined or `radix_4_burst_io`.

The `hls_fft.h` header file defines a struct `hls::ip_fft::params_t` which can be used to set default values for the static parameters. If the default values are to be used, the parameterization struct can be used directly with the FFT function.

```
hls::fft<hls::ip_fft::params_t >
    (xn1, xk1, &fft_status1, &fft_config1);
```

A more typical use is to change some of the parameters to non-default values. This is performed by creating a new user-defined parameterization struct based on the default parameterization struct and changing some of the default values.

In the following example, a new user struct `my_fft_config` is defined with a new value for the output ordering (changed to `natural_order`). All other static parameters to the FFT use the default values.

```
struct my_fft_config : hls::ip_fft::params_t {
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
};

hls::fft<my_fft_config >
    (xn1, xk1, &fft_status1, &fft_config1);
```

The values used for the parameterization struct `hls::ip_fft::params_t` are explained in [FFT Struct Parameters](#). The default values for the parameters and a list of possible values are provided in [FFT Struct Parameter Values](#).



RECOMMENDED: Xilinx highly recommends that you review the Fast Fourier Transform LogiCORE IP Product Guide ([PG109](#)) for details on the parameters and the implication for their settings.

FFT Struct Parameters

Table 43: FFT Struct Parameters

Parameter	Description
<code>input_width</code>	Data input port width.
<code>output_width</code>	Data output port width.
<code>status_width</code>	Output status port width.
<code>config_width</code>	Input configuration port width.
<code>max_nfft</code>	The size of the FFT data set is specified as <code>1 << max_nfft</code> .
<code>has_nfft</code>	Determines if the size of the FFT can be runtime configurable.
<code>channels</code>	Number of channels.

Table 43: FFT Struct Parameters (cont'd)

Parameter	Description
arch_opt	The implementation architecture.
phase_factor_width	Configure the internal phase factor precision.
ordering_opt	The output ordering mode.
ovflo	Enable overflow mode.
scaling_opt	Define the scaling options.
rounding_opt	Define the rounding modes.
mem_data	Specify using block or distributed RAM for data memory.
mem_phase_factors	Specify using block or distributed RAM for phase factors memory.
mem_reorder	Specify using block or distributed RAM for output reorder memory.
stages_block_ram	Defines the number of block RAM stages used in the implementation.
mem_hybrid	When block RAMs are specified for data, phase factor, or reorder buffer, mem_hybrid specifies where or not to use a hybrid of block and distributed RAMs to reduce block RAM count in certain configurations.
complex_mult_type	Defines the types of multiplier to use for complex multiplications.
butterfly_type	Defines the implementation used for the FFT butterfly.

When specifying parameter values which are not integer or boolean, the HLS FFT namespace should be used.

For example, the possible values for parameter `butterfly_type` in the following table are `use_luts` and `use_xtremedsp_slices`. The values used in the C program should be `butterfly_type = hls::ip_fft::use_luts` and `butterfly_type = hls::ip_fft::use_xtremedsp_slices`.

FFT Struct Parameter Values

The following table covers all features and functionality of the FFT IP. Features and functionality not described in this table are not supported in the Vitis HLS implementation.

Table 44: FFT Struct Parameter Values

Parameter	C Type	Default Value	Valid Values
input_width	unsigned	16	8-34
output_width	unsigned	16	input_width to (input_width + max_nfft + 1)
status_width	unsigned	8	Depends on FFT configuration
config_width	unsigned	16	Depends on FFT configuration
max_nfft	unsigned	10	3-16
has_nfft	bool	false	True, False
channels	unsigned	1	1-12

Table 44: FFT Struct Parameter Values (cont'd)

Parameter	C Type	Default Value	Valid Values
arch_opt	unsigned	pipelined_streaming_io	automatically_select pipelined_streaming_io radix_4_burst_io radix_2_burst_io radix_2_lite_burst_io
phase_factor_width	unsigned	16	8-34
ordering_opt	unsigned	bit_reversed_order	bit_reversed_order natural_order
ovflo	bool	true	false true
scaling_opt	unsigned	scaled	scaled unscaled block_floating_point
rounding_opt	unsigned	truncation	truncation convergent_rounding
mem_data	unsigned	block_ram	block_ram distributed_ram
mem_phase_factors	unsigned	block_ram	block_ram distributed_ram
mem_reorder	unsigned	block_ram	block_ram distributed_ram
stages_block_ram	unsigned	(max_nfft < 10) ? 0 : (max_nfft - 9)	0-11
mem_hybrid	bool	false	false true
complex_mult_type	unsigned	use_mults_resources	use_luts use_mults_resources use_mults_performance
butterfly_type	unsigned	use_luts	use_luts use_xtremedsp_slices

FFT Runtime Configuration and Status

The FFT supports runtime configuration and runtime status monitoring through the configuration and status ports. These ports are defined as arguments to the FFT function, shown here as variables `fft_status1` and `fft_config1`:

```
hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);
```

The runtime configuration and status can be accessed using the predefined structs from the FFT C library:

- `hls::ip_fft::config_t<param1>`
- `hls::ip_fft::status_t<param1>`

Note: In both cases, the struct requires the name of the static parameterization struct, shown in these examples as param1. Refer to the previous section for details on defining the static parameterization struct.

The runtime configuration struct allows the following actions to be performed in the C code:

- Set the FFT length, if runtime configuration is enabled
- Set the FFT direction as forward or inverse
- Set the scaling schedule

The FFT length can be set as follows:

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
// Set FFT length to 512 => log2(512) =>9
fft_config1->setNfft(9);
```



IMPORTANT! The length specified during runtime cannot exceed the size defined by `max_nfft` in the static configuration.

The FFT direction can be set as follows:

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
// Forward FFT
fft_config1->setDir(1);
// Inverse FFT
fft_config1->setDir(0);
```

The FFT scaling schedule can be set as follows:

```
typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
fft_config1->setSch(0x2AB);
```

The output status port can be accessed using the pre-defined struct to determine:

- If any overflow occurred during the FFT
- The value of the block exponent

The FFT overflow mode can be checked as follows:

```
typedef hls::ip_fft::status_t<param1> status_t;
status_t fft_status1;
// Check the overflow flag
bool *ovflo = fft_status1->getOvflo();
```



IMPORTANT! After each transaction completes, check the overflow status to confirm the correct operation of the FFT.

And the block exponent value can be obtained using:

```
typedef hls::ip_fft::status_t<param1> status_t;
status_t fft_status1;
// Obtain the block exponent
unsigned int *blk_exp = fft_status1-> getBlkExp();
```

Using the FFT Function

The FFT function is defined in the HLS namespace and can be called as follows:

```
hls::fft<STATIC_PARAM> (
INPUT_DATA_ARRAY,
OUTPUT_DATA_ARRAY,
OUTPUT_STATUS,
INPUT_RUN_TIME_CONFIGURATION);
```

The STATIC_PARAM is the static parameterization struct that defines the static parameters for the FFT.

Both the input and output data are supplied to the function as arrays (INPUT_DATA_ARRAY and OUTPUT_DATA_ARRAY). In the final implementation, the ports on the FFT RTL block will be implemented as AXI4-Stream ports. Xilinx recommends always using the FFT function in a region using dataflow optimization (set_directive_dataflow), because this ensures the arrays are implemented as streaming arrays. An alternative is to specify both arrays as streaming using the set_directive_stream command.



IMPORTANT! The FFT cannot be used in a region which is pipelined. If high-performance operation is required, pipeline the loops or functions before and after the FFT then use dataflow optimization on all loops and functions in the region.

The data types for the arrays can be float or ap_fixed.

```
typedef float data_t;
complex<data_t> xn[FFT_LENGTH];
complex<data_t> xk[FFT_LENGTH];
```

To use fixed-point data types, the Vitis HLS arbitrary precision type ap_fixed should be used.

```
#include "ap_fixed.h"
typedef ap_fixed<FFT_INPUT_WIDTH,1> data_in_t;
typedef ap_fixed<FFT_OUTPUT_WIDTH,FFT_OUTPUT_WIDTH-FFT_INPUT_WIDTH+1>
data_out_t;
#include <complex>
typedef hls::x_complex<data_in_t> cmpxData;
typedef hls::x_complex<data_out_t> cmpxDataOut;
```

In both cases, the FFT should be parameterized with the same correct data sizes. In the case of floating point data, the data widths will always be 32-bit and any other specified size will be considered invalid.



IMPORTANT! The input and output width of the FFT can be configured to any arbitrary value within the supported range. The variables which connect to the input and output parameters must be defined in increments of 8-bit. For example, if the output width is configured as 33-bit, the output variable must be defined as a 40-bit variable.

The multichannel functionality of the FFT can be used by using two-dimensional arrays for the input and output data. In this case, the array data should be configured with the first dimension representing each channel and the second dimension representing the FFT data.

```
typedef float data_t;
static complex<data_t> xn[CHANNEL][FFT_LENGTH];
static complex<data_t> xk[CHANELL][FFT_LENGTH];
```

The FFT core consumes and produces data as interleaved channels (for example, ch0-data0, ch1-data0, ch2-data0, etc, ch0-data1, ch1-data1, ch2-data2, etc.). Therefore, to stream the input or output arrays of the FFT using the same sequential order that the data was read or written, you must fill or empty the two-dimensional arrays for multiple channels by iterating through the channel index first, as shown in the following example:

```
cmplxData in_fft[FFT_CHANNELS][FFT_LENGTH];
cmplxData out_fft[FFT_CHANNELS][FFT_LENGTH];

// Write to FFT Input Array
for (unsigned i = 0; i < FFT_LENGTH; i++) {
    for (unsigned j = 0; j < FFT_CHANNELS; ++j) {
        in_fft[j][i] = in.read().data;
    }
}

// Read from FFT Output Array
for (unsigned i = 0; i < FFT_LENGTH; i++) {
    for (unsigned j = 0; j < FFT_CHANNELS; ++j) {
        out.data = out_fft[j][i];
    }
}
```

Design examples using the FFT C library are provided in the Vitis HLS examples and can be accessed using menu option **Help**→**Welcome**→**Open Example Project**→**Design Examples**→**FFT**.

FIR Filter IP Library

The Xilinx FIR IP block can be called within a C++ design using the library `hls_fir.h`. This section explains how the FIR can be configured in your C++ code.



RECOMMENDED: Xilinx highly recommends that you review the FIR Compiler LogiCORE IP Product Guide ([PG149](#)) for information on how to implement and use the features of the IP.

To use the FIR in your C++ code:

1. Include the `hls_fir.h` library in the code.
2. Set the static parameters using the predefined struct `hls::ip_fir::params_t`.
3. Call the FIR function.
4. Optionally, define a runtime input configuration to modify some parameters dynamically.

The following code examples provide a summary of how each of these steps is performed. Each step is discussed in more detail below.

First, include the FIR library in the source code. This header file resides in the include directory in the Vitis HLS installation area. This directory is automatically searched when Vitis HLS executes. There is no need to specify the path to this directory if compiling inside Vitis HLS.

```
#include "hls_fir.h"
```

Define the static parameters of the FIR. This includes such static attributes such as the input width, the coefficients, the filter rate (`single`, `decimation`, `hilbert`). The FIR library includes a parameterization struct `hls::ip_fir::params_t` which can be used to initialize all static parameters with default values.

In this example, the coefficients are defined as residing in array `coeff_vec` and the default values for the number of coefficients, the input width and the quantization mode are over-ridden using a user-defined struct `myconfig` based on the predefined struct.

```
struct myconfig : hls::ip_fir::params_t {
    static const double coeff_vec[sg_fir_srrc_coeffs_len];
    static const unsigned num_coeffs = sg_fir_srrc_coeffs_len;
    static const unsigned input_width = INPUT_WIDTH;
    static const unsigned quantization = hls::ip_fir::quantize_only;
};
```

Create an instance of the FIR function using the HLS namespace with the defined static parameters (`myconfig` in this example) and then call the function with the `run` method to execute the function. The function arguments are, in order, input data and output data.

```
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out);
```

Optionally, a runtime input configuration can be used. In some modes of the FIR, the data on this input determines how the coefficients are used during interleaved channels or when coefficient reloading is required. This configuration can be dynamic and is therefore defined as a variable. For a complete description of which modes require this input configuration, refer to the *FIR Compiler LogiCORE IP Product Guide* (PG149).

When the runtime input configuration is used, the FIR function is called with three arguments: input data, output data and input configuration.

```
// Define the configuration type
typedef ap_uint<8> config_t;
// Define the configuration variable
config_t fir_config = 8;
// Use the configuration in the FFT
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out, &fir_config);
```

FIR Static Parameters

The static parameters of the FIR define how the FIR IP is parameterized and specifies non-dynamic items such as the input and output widths, the number of fractional bits, the coefficient values, the interpolation and decimation rates. Most of these configurations have default values: there are no default values for the coefficients.

The `hls_fir.h` header file defines a `struct hls::ip_fir::params_t` that can be used to set the default values for most of the static parameters.

IMPORTANT! *There are no defaults defined for the coefficients. Therefore, Xilinx does not recommend using the pre-defined struct to directly initialize the FIR. A new user defined struct which specifies the coefficients should always be used to perform the static parameterization.*

In this example, a new user `struct my_config` is defined and with a new value for the coefficients. The coefficients are specified as residing in array `coeff_vec`. All other parameters to the FIR use the default values.

```
struct myconfig : hls::ip_fir::params_t {
    static const double coeff_vec[sg_fir_srrc_coeffs_len];
};
static hls::FIR<myconfig> fir1;
fir1.run(fir_in, fir_out);
```

[FIR Static Parameters](#) describes the parameters used for the parametrization struct `hls::ip_fir::params_t`. [FIR Struct Parameter Values](#) provides the default values for the parameters and a list of possible values.

RECOMMENDED: *Xilinx highly recommends that you refer to the FIR Compiler LogiCORE IP Product Guide ([PG149](#)) for details on the parameters and the implication for their settings.*

FIR Struct Parameters

Table 45: FIR Struct Parameters

Parameter	Description
<code>input_width</code>	Data input port width
<code>input_fractional_bits</code>	Number of fractional bits on the input port

Table 45: FIR Struct Parameters (cont'd)

Parameter	Description
output_width	Data output port width
output_fractional_bits	Number of fractional bits on the output port
coeff_width	Bit-width of the coefficients
coeff_fractional_bits	Number of fractional bits in the coefficients
num_coeffs	Number of coefficients
coeff_sets	Number of coefficient sets
input_length	Number of samples in the input data
output_length	Number of samples in the output data
num_channels	Specify the number of channels of data to process
total_num_coeff	Total number of coefficients
coeff_vec[total_num_coeff]	The coefficient array
filter_type	The type implementation used for the filter
rate_change	Specifies integer or fractional rate changes
interp_rate	The interpolation rate
decim_rate	The decimation rate
zero_pack_factor	Number of zero coefficients used in interpolation
rate_specification	Specify the rate as frequency or period
hardware_oversampling_rate	Specify the rate of over-sampling
sample_period	The hardware oversample period
sample_frequency	The hardware oversample frequency
quantization	The quantization method to be used
best_precision	Enable or disable the best precision
coeff_structure	The type of coefficient structure to be used
output_rounding_mode	Type of rounding used on the output
filter_arch	Selects a systolic or transposed architecture
optimization_goal	Specify a speed or area goal for optimization
inter_column_pipe_length	The pipeline length required between DSP columns
column_config	Specifies the number of DSP module columns
config_method	Specifies how the DSP module columns are configured
coeff_padding	Number of zero padding added to the front of the filter

When specifying parameter values that are not integer or boolean, the HLS FIR namespace should be used.

For example the possible values for `rate_change` are shown in the following table to be `integer` and `fixed_fractional`. The values used in the C program should be
`rate_change = hls::ip_fir::integer` and `rate_change = hls::ip_fir::fixed_fractional`.

FIR Struct Parameter Values

The following table covers all features and functionality of the FIR IP. Features and functionality not described in this table are not supported in the Vitis HLS implementation.

Table 46: FIR Struct Parameter Values

Parameter	C Type	Default Value	Valid Values
input_width	unsigned	16	No limitation
input_fractional_bits	unsigned	0	Limited by size of input_width
output_width	unsigned	24	No limitation
output_fractional_bits	unsigned	0	Limited by size of output_width
coeff_width	unsigned	16	No limitation
coeff_fractional_bits	unsigned	0	Limited by size of coeff_width
num_coeffs	bool	21	Full
coeff_sets	unsigned	1	1-1024
input_length	unsigned	21	No limitation
output_length	unsigned	21	No limitation
num_channels	unsigned	1	1-1024
total_num_coeff	unsigned	21	num_coeffs * coeff_sets
coeff_vec[total_num_coeff]	double array	None	Not applicable
filter_type	unsigned	single_rate	single_rate, interpolation, decimation, hilbert_filter, interpolated
rate_change	unsigned	integer	integer, fixed_fractional
interp_rate	unsigned	1	1-1024
decim_rate	unsigned	1	1-1024
zero_pack_factor	unsigned	1	1-8
rate_specification	unsigned	period	frequency, period
hardware_oversampling_rate	unsigned	1	No Limitation
sample_period	bool	1	No Limitation
sample_frequency	unsigned	0.001	No Limitation
quantization	unsigned	integer_coefficients	integer_coefficients, quantize_only, maximize_dynamic_range
best_precision	unsigned	false	false true
coeff_structure	unsigned	non_symmetric	inferred, non_symmetric, symmetric, negative_symmetric, half_band, hilbert
output_rounding_mode	unsigned	full_precision	full_precision, truncate_lsbs, non_symmetric_rounding_down, non_symmetric_rounding_up, symmetric_rounding_to_zero, symmetric_rounding_to_infinity, convergent_rounding_to_even, convergent_rounding_to_odd

Table 46: FIR Struct Parameter Values (cont'd)

Parameter	C Type	Default Value	Valid Values
filter_arch	unsigned	systolic_multiply_accumulate	systolic_multiply_accumulate, transpose_multiply_accumulate
optimization_goal	unsigned	area	area, speed
inter_column_pipe_length	unsigned	4	1-16
column_config	unsigned	1	Limited by number of DSP macrocells used
config_method	unsigned	single	single, by_channel
coeff_padding	bool	false	false true

Using the FIR Function

The FIR function is defined in the HLS namespace and can be called as follows:

```
// Create an instance of the FIR
static hls::FIR<STATIC_PARAM> fir1;
// Execute the FIR instance fir1
fir1.run(INPUT_DATA_ARRAY, OUTPUT_DATA_ARRAY);
```

The STATIC_PARAM is the static parameterization struct that defines most static parameters for the FIR.

Both the input and output data are supplied to the function as arrays (INPUT_DATA_ARRAY and OUTPUT_DATA_ARRAY). In the final implementation, these ports on the FIR IP will be implemented as AXI4-Stream ports. Xilinx recommends always using the FIR function in a region using the dataflow optimization (`set_directive_dataflow`), because this ensures the arrays are implemented as streaming arrays. An alternative is to specify both arrays as streaming using the `set_directive_stream` command.



IMPORTANT! The FIR cannot be used in a region which is pipelined. If high-performance operation is required, pipeline the loops or functions before and after the FIR then use dataflow optimization on all loops and functions in the region.

The multichannel functionality of the FIR is supported through interleaving the data in a single input and single output array.

- The size of the input array should be large enough to accommodate all samples:
`num_channels * input_length`.
- The output array size should be specified to contain all output samples: `num_channels * output_length`.

The following code example demonstrates, for two channels, how the data is interleaved. In this example, the top-level function has two channels of input data (`din_i`, `din_q`) and two channels of output data (`dout_i`, `dout_q`). Two functions, at the front-end (fe) and back-end (be) are used to correctly order the data in the FIR input array and extract it from the FIR output array.

```

void dummy_fe(din_t din_i[LENGTH], din_t din_q[LENGTH], din_t
out[FIR_LENGTH]) {
    for (unsigned i = 0; i < LENGTH; ++i) {
        out[2*i] = din_i[i];
        out[2*i + 1] = din_q[i];
    }
}
void dummy_be(dout_t in[FIR_LENGTH], dout_t dout_i[LENGTH], dout_t
dout_q[LENGTH]) {
    for(unsigned i = 0; i < LENGTH; ++i) {
        dout_i[i] = in[2*i];
        dout_q[i] = in[2*i+1];
    }
}
void fir_top(din_t din_i[LENGTH], din_t din_q[LENGTH],
             dout_t dout_i[LENGTH], dout_t dout_q[LENGTH]) {

    din_t fir_in[FIR_LENGTH];
    dout_t fir_out[FIR_LENGTH];
    static hls::FIR<myconfig> fir1;

    dummy_fe(din_i, din_q, fir_in);
    fir1.run(fir_in, fir_out);
    dummy_be(fir_out, dout_i, dout_q);
}

```

Optional FIR Runtime Configuration

In some modes of operation, the FIR requires an additional input to configure how the coefficients are used. For a complete description of which modes require this input configuration, refer to the *FIR Compiler LogiCORE IP Product Guide (PG149)*.

This input configuration can be performed in the C code using a standard `ap_int.h` 8-bit data type. In this example, the header file `fir_top.h` specifies the use of the FIR and `ap_fixed` libraries, defines a number of the design parameter values and then defines some fixed-point types based on these:

```

#include "ap_fixed.h"
#include "hls_fir.h"

const unsigned FIR_LENGTH      = 21;
const unsigned INPUT_WIDTH     = 16;
const unsigned INPUT_FRACTIONAL_BITS = 0;
const unsigned OUTPUT_WIDTH    = 24;
const unsigned OUTPUT_FRACTIONAL_BITS = 0;
const unsigned COEFF_WIDTH     = 16;
const unsigned COEFF_FRACTIONAL_BITS = 0;
const unsigned COEFF_NUM       = 7;
const unsigned COEFF_SETS      = 3;
const unsigned INPUT_LENGTH    = FIR_LENGTH;

```

```

const unsigned OUTPUT_LENGTH = FIR_LENGTH;
const unsigned CHAN_NUM = 1;
typedef ap_fixed<INPUT_WIDTH, INPUT_WIDTH - INPUT_FRACTIONAL_BITS> s_data_t;
typedef ap_fixed<OUTPUT_WIDTH, OUTPUT_WIDTH - OUTPUT_FRACTIONAL_BITS>
m_data_t;
typedef ap_uint<8> config_t;

```

In the top-level code, the information in the header file is included, the static parameterization struct is created using the same constant values used to specify the bit-widths, ensuring the C code and FIR configuration match, and the coefficients are specified. At the top-level, an input configuration, defined in the header file as 8-bit data, is passed into the FIR.

```

#include "fir_top.h"

struct param1 : hls::ip_fir::params_t {
    static const double coeff_vec[total_num_coeff];
    static const unsigned input_length = INPUT_LENGTH;
    static const unsigned output_length = OUTPUT_LENGTH;
    static const unsigned num_coeffs = COEFF_NUM;
    static const unsigned coeff_sets = COEFF_SETS;
};

const double param1::coeff_vec[total_num_coeff] =
{6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6};

void dummy_fe(s_data_t in[INPUT_LENGTH], s_data_t out[INPUT_LENGTH],
              config_t* config_in, config_t* config_out)
{
    *config_out = *config_in;
    for(unsigned i = 0; i < INPUT_LENGTH; ++i)
        out[i] = in[i];
}

void dummy_be(m_data_t in[OUTPUT_LENGTH], m_data_t out[OUTPUT_LENGTH])
{
    for(unsigned i = 0; i < OUTPUT_LENGTH; ++i)
        out[i] = in[i];
}

// DUT
void fir_top(s_data_t in[INPUT_LENGTH],
             m_data_t out[OUTPUT_LENGTH],
             config_t* config)
{
    s_data_t fir_in[INPUT_LENGTH];
    m_data_t fir_out[OUTPUT_LENGTH];
    config_t fir_config;
    // Create struct for config
    static hls::FIR<param1> fir1;

    //=====
    // Dataflow process
    dummy_fe(in, fir_in, config, &fir_config);
    fir1.run(fir_in, fir_out, &fir_config);
    dummy_be(fir_out, out);
    //=====
}

```

Design examples using the FIR C library are provided in the Vitis HLS examples and can be accessed using menu option **Help**→**Welcome**→**Open Example Project**→**Design Examples**→**FIR**.

DDS IP Library

You can use the Xilinx Direct Digital Synthesizer (DDS) IP block within a C++ design using the `hls_dds.h` library. This section explains how to configure DDS IP in your C++ code.



RECOMMENDED: Xilinx highly recommends that you review the DDS Compiler LogiCORE IP Product Guide ([PG141](#)) for information on how to implement and use the features of the IP.



IMPORTANT! The C IP implementation of the DDS IP core supports the `fixed` mode for the `Phase_Increment` and `Phase_Offset` parameters and supports the `none` mode for `Phase_Offset`, but it does not support `programmable` and `streaming` modes for these parameters.

To use the DDS in the C++ code:

1. Include the `hls_dds.h` library in the code.
2. Set the default parameters using the pre-defined struct `hls::ip_dds::params_t`.
3. Call the DDS function.

First, include the DDS library in the source code. This header file resides in the include directory in the Vitis HLS installation area, which is automatically searched when Vitis HLS executes.

```
#include "hls_dds.h"
```

Define the static parameters of the DDS. For example, define the phase width, clock rate, and phase and increment offsets. The DDS C library includes a parameterization struct `hls::ip_dds::params_t`, which is used to initialize all static parameters with default values. By redefining any of the values in this struct, you can customize the implementation.

The following example shows how to override the default values for the phase width, clock rate, phase offset, and the number of channels using a user-defined struct `param1`, which is based on the existing predefined struct `hls::ip_dds::params_t`:

```
struct param1 : hls::ip_dds::params_t {
    static const unsigned Phase_Width = PHASEWIDTH;
    static const double DDS_Clock_Rate = 25.0;
    static const double PINC[16];
    static const double POFF[16];
};
```

Create an instance of the DDS function using the HLS namespace with the defined static parameters (for example, param1). Then, call the function with the run method to execute the function. Following are the data and phase function arguments shown in order:

```
static hls::DDS<config1> dds1;
dds1.run(data_channel, phase_channel);
```

DDS Static Parameters

The static parameters of the DDS define how to configure the DDS, such as the clock rate, phase interval, and modes. The `hls_dds.h` header file defines an `hls::ip_dds::params_t` struct, which sets the default values for the static parameters. To use the default values, you can use the parameterization struct directly with the DDS function.

```
static hls::DDS< hls::ip_dds::params_t > dds1;
dds1.run(data_channel, phase_channel);
```

The following table describes the parameters for the `hls::ip_dds::params_t` parameterization struct.



RECOMMENDED: Xilinx highly recommends that you review the DDS Compiler LogiCORE IP Product Guide ([PG141](#)) for details on the parameters and values.

Table 47: DDS Struct Parameters

Parameter	Description
<code>DDS_Clock_Rate</code>	Specifies the clock rate for the DDS output.
<code>Channels</code>	Specifies the number of channels. The DDS and phase generator can support up to 16 channels. The channels are time-multiplexed, which reduces the effective clock frequency per channel.
<code>Mode_of_Operation</code>	Specifies one of the following operation modes: Standard mode for use when the accumulated phase can be truncated before it is used to access the SIN/COS LUT. Rasterized mode for use when the desired frequencies and system clock are related by a rational fraction.
<code>Modulus</code>	Describes the relationship between the system clock frequency and the desired frequencies. Use this parameter in rasterized mode only.
<code>Spurious_Free_Dynamic_Range</code>	Specifies the targeted purity of the tone produced by the DDS.
<code>Frequency_Resolution</code>	Specifies the minimum frequency resolution in Hz and determines the Phase Width used by the phase accumulator, including associated phase increment (PINC) and phase offset (POFF) values.
<code>Noise_Shaping</code>	Controls whether to use phase truncation, dithering, or Taylor series correction.

Table 47: DDS Struct Parameters (cont'd)

Parameter	Description
Phase_Width	Sets the width of the following: <ul style="list-style-type: none"> • PHASE_OUT field within <code>m_axis_phase_tdata</code> • Phase field within <code>s_axis_phase_tdata</code> when the DDS is configured to be a SIN/COS LUT only • Phase accumulator • Associated phase increment and offset registers • Phase field in <code>s_axis_config_tdata</code> For rasterized mode, the phase width is fixed as the number of bits required to describe the valid input range [0, Modulus-1], that is, $\log_2 (\text{Modulus}-1)$ rounded up.
Output_Width	Sets the width of SINE and COSINE fields within <code>m_axis_data_tdata</code> . The SFDR provided by this parameter depends on the selected Noise Shaping option.
Phase_Increment	Selects the phase increment value.
Phase_Offset	Selects the phase offset value.
Output_Selection	Sets the output selection to SINE , COSINE , or both in the <code>m_axis_data_tdata</code> bus.
Negative_Sine	Negates the SINE field at runtime.
Negative_Cosine	Negates the COSINE field at runtime.
Amplitude_Mode	Sets the amplitude to full range or unit circle.
Memory_Type	Controls the implementation of the SIN/COS LUT.
Optimization_Goal	Controls whether the implementation decisions target highest speed or lowest resource.
DSP48_Use	Controls the implementation of the phase accumulator and addition stages for phase offset, dither noise addition, or both.
Latency_Configuration	Sets the latency of the core to the optimum value based upon the Optimization Goal.
Latency	Specifies the manual latency value.
Output_Form	Sets the output form to two's complement or to sign and magnitude. In general, the output of SINE and COSINE is in two's complement form. However, when quadrant symmetry is used, the output form can be changed to sign and magnitude.
PINC[XIP_DDS_CHANNELS_MAX]	Sets the values for the phase increment for each output channel.
POFF[XIP_DDS_CHANNELS_MAX]	Sets the values for the phase offset for each output channel.

DDS Struct Parameter Values

The following table shows the possible values for the `hls::ip_dds::params_t` parameterization struct parameters.

Table 48: DDS Struct Parameter Values

Parameter	C Type	Default Value	Valid Values
DDS_Clock_Rate	double	20.0	Any double value

Table 48: DDS Struct Parameter Values (cont'd)

Parameter	C Type	Default Value	Valid Values
Channels	unsigned	1	1 to 16
Mode_of_Operation	unsigned	XIP_DDS_MOO_CONVENTIONAL	XIP_DDS_MOO_CONVENTIONAL truncates the accumulated phase. XIP_DDS_MOO_RASTERIZED selects rasterized mode.
Modulus	unsigned	200	129 to 256
Spurious_Free_Dynamic_Range	double	20.0	18.0 to 150.0
Frequency_Resolution	double	10.0	0.000000001 to 125000000
Noise_Shaping	unsigned	XIP_DDS_NS_NONE	XIP_DDS_NS_NONE produces phase truncation DDS. XIP_DDS_NS_DITHER uses phase dither to improve SFDR at the expense of increased noise floor. XIP_DDS_NS_TAYLOR interpolates sine/cosine values using the otherwise discarded bits from phase truncation XIP_DDS_NS_AUTO automatically determines noise-shaping.
Phase_Width	unsigned	16	Must be an integer multiple of 8
Output_Width	unsigned	16	Must be an integer multiple of 8
Phase_Increment	unsigned	XIP_DDS_PINCPOFF_FIXED	XIP_DDS_PINCPOFF_FIXED fixes PINC at generation time, and PINC cannot be changed at runtime. This is the only value supported.
Phase_Offset	unsigned	XIP_DDS_PINCPOFF_NONE	XIP_DDS_PINCPOFF_NONE does not generate phase offset. XIP_DDS_PINCPOFF_FIXED fixes POFF at generation time, and POFF cannot be changed at runtime.
Output_Selection	unsigned	XIP_DDS_OUT_SIN_AND_COS	XIP_DDS_OUT_SIN_ONLY produces sine output only. XIP_DDS_OUT_COS_ONLY produces cosine output only. XIP_DDS_OUT_SIN_AND_COS produces both sin and cosine output.
Negative_Sine	unsigned	XIP_DDS_ABSENT	XIP_DDS_ABSENT produces standard sine wave. XIP_DDS_PRESENT negates sine wave.
Negative_Cosine	bool	XIP_DDS_ABSENT	XIP_DDS_ABSENT produces standard sine wave. XIP_DDS_PRESENT negates sine wave.

Table 48: DDS Struct Parameter Values (cont'd)

Parameter	C Type	Default Value	Valid Values
Amplitude_Mode	unsigned	XIP_DDS_FULL_RANGE	XIP_DDS_FULL_RANGE normalizes amplitude to the output width with the binary point in the first place. For example, an 8-bit output has a binary amplitude of 100000000 - 10 giving values between 01111110 and 11111110, which corresponds to just less than 1 and just more than -1, respectively. XIP_DDS_UNIT_CIRCLE normalizes amplitude to half full range, that is, values range from 01000 .. (+0.5). to 110000 .. (-0.5).
Memory_Type	unsigned	XIP_DDS_MEM_AUTO	XIP_DDS_MEM_AUTO selects distributed ROM for small cases where the table can be contained in a single layer of memory and selects block ROM for larger cases. XIP_DDS_MEM_BLOCK always uses block RAM. XIP_DDS_MEM_DIST always uses distributed RAM.
Optimization_Goal	unsigned	XIP_DDS_OPTGOAL_AUTO	XIP_DDS_OPTGOAL_AUTO automatically selects the optimization goal. XIP_DDS_OPTGOAL_AREA optimizes for area. XIP_DDS_OPTGOAL_SPEED optimizes for performance.
DSP48_Use	unsigned	XIP_DDS_DSP_MIN	XIP_DDS_DSP_MIN implements the phase accumulator and the stages for phase offset, dither noise addition, or both in FPGA logic. XIP_DDS_DSP_MAX implements the phase accumulator and the phase offset, dither noise addition, or both using DSP slices. In the case of single channel, the DSP slice can also provide the register to store programmable phase increment, phase offset, or both and thereby, save further fabric resources.
Latency_Configuration	unsigned	XIP_DDS_LATENCY_AUTO	XIP_DDS_LATENCY_AUTO automatically determines he latency. XIP_DDS_LATENCY_MANUAL manually specifies the latency using the Latency option.
Latency	unsigned	5	Any value

Table 48: DDS Struct Parameter Values (cont'd)

Parameter	C Type	Default Value	Valid Values
Output_Form	unsigned	XIP_DDS_OUTPUT_TWOS	XIP_DDS_OUTPUT_TWOS outputs two's complement. XIP_DDS_OUTPUT_SIGN_MAG outputs signed magnitude.
PINC[XIP_DDS_CHANNELS_MAX]	unsigned array	{0}	Any value for the phase increment for each channel
POFF[XIP_DDS_CHANNELS_MAX]	unsigned array	{0}	Any value for the phase offset for each channel

SRL IP Library

C code is written to satisfy several different requirements: reuse, readability, and performance. Until now, it is unlikely that the C code was written to result in the most ideal hardware after high-level synthesis.

Like the requirements for reuse, readability, and performance, certain coding techniques or pre-defined constructs can ensure that the synthesis output results in more optimal hardware or to better model hardware in C for easier validation of the algorithm.

Mapping Directly into SRL Resources

Many C algorithms sequentially shift data through arrays. They add a new value to the start of the array, shift the existing data through array, and drop the oldest data value. This operation is implemented in hardware as a shift register.

This most common way to implement a shift register from C into hardware is to completely partition the array into individual elements, and allow the data dependencies between the elements in the RTL to imply a shift register.

Logic synthesis typically implements the RTL shift register into a Xilinx SRL resource, which efficiently implements shift registers. The issue is that sometimes logic synthesis does not implement the RTL shift register using an SRL component:

- When data is accessed in the middle of the shift register, logic synthesis cannot directly infer an SRL.
- Sometimes, even when the SRL is ideal, logic synthesis may implement the shift-resister in flip-flops, due to other factors. (Logic synthesis is also a complex process).

Vitis HLS provides a C++ class (`ap_shift_reg`) to ensure that the shift register defined in the C code is always implemented using an SRL resource. The `ap_shift_reg` class has two methods to perform the various read and write accesses supported by an SRL component.

Read from the Shifter

The read method allows a specified location to be read from the shifter register.

The `ap_shift_reg.h` header file that defines the `ap_shift_reg` class is also included with Vitis HLS as a standalone package. You have the right to use it in your own source code. The package `xilinx_hls_lib_<release_number>.tgz` is located in the `include` directory in the Vitis HLS installation area.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 2 of Sreg into var1
var1 = Sreg.read(2);
```

Read, Write, and Shift Data

A shift method allows a read, write, and shift operation to be performed.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1;

// Read location 3 of Sreg into var1
// THEN shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3);
```

Read, Write, and Enable-Shift

The shift method also supports an enabled input, allowing the shift process to be controlled and enabled by a variable.

```
// Include the Class
#include "ap_shift_reg.h"

// Define a variable of type ap_shift_reg<type, depth>
// - Sreg must use the static qualifier
// - Sreg will hold integer data types
// - Sreg will hold 4 data values
static ap_shift_reg<int, 4> Sreg;
int var1, In1;
```

```
bool En;  
  
// Read location 3 of Sreg into var1  
// THEN if En=1  
// Shift all values up one and load In1 into location 0  
var1 = Sreg.shift(In1,3,En);
```

When using the `ap_shift_reg` class, Vitis HLS creates a unique RTL component for each shifter. When logic synthesis is performed, this component is synthesized into an SRL resource.