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

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

void hier_func4(din_t A, din_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) {

#ifdef 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;
#ifdef __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 an 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 ports 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	1. Issues an error message. 2. Halts synthesis.
Indeterminate	1. Issues a warning. 2. 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 [XFORM-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| DSP48E|   FF   |   LUT  |
+-----------------+---------+-------+--------+--------+
|Expression       |        -|      -|       0|      13|
|FIFO             |        -|      -|       -|       -|
|Instance         |        -|      1|    4764|    4560|
|Memory           |        -|      -|       -|       -|
|Multiplexer      |        -|      -|       -|       -|
|Register         |        -|      -|       6|       -|
+-----------------+---------+-------+--------+--------+
|Total            |        0|      1|    4770|    4573|
+-----------------+---------+-------+--------+--------+
|Available        |      650|    600|  202800|  101400|
+-----------------+---------+-------+--------+--------+
|Utilization (%)  |        0|   ~0  |       2|       4|
+-----------------+---------+-------+--------+--------+
```

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

Overview of Arbitrary Precision Integer Data Types

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

Table 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	The number of bits used to represent the integer value, that is, the number of integer bits to the *left* of the binary point. When this value is negative, it represents the number of *implicit* sign bits (for signed representation), or the number of *implicit* zero bits (for unsigned representation) to the *right* of the binary point. For example,
	`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`

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).

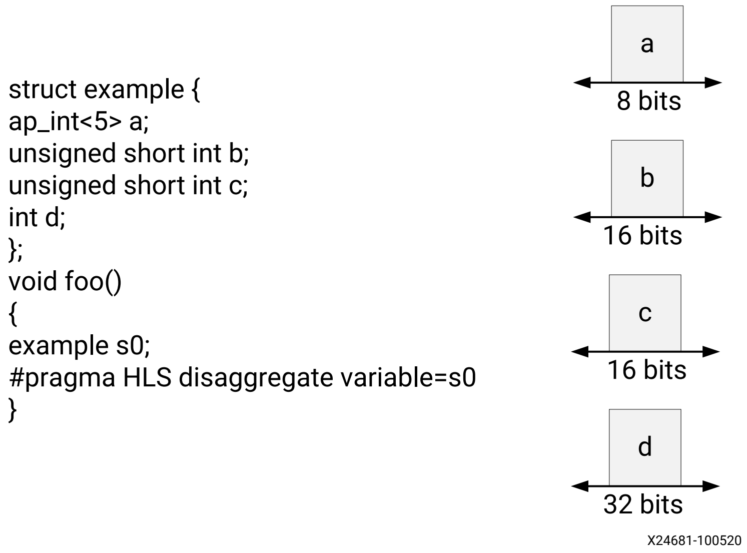
Struct Padding and Alignment

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
}
```

- **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.

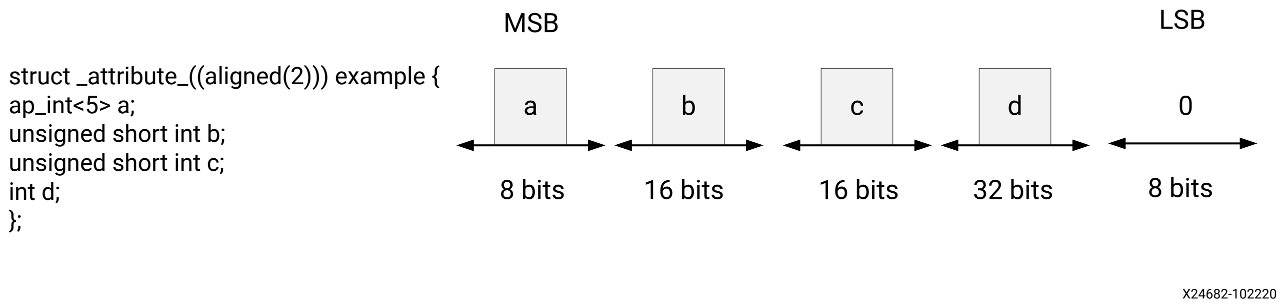
Data types in struct with custom data widths, such as `ap_int`, are allocated with sizes which are powers of 2. Vitis HLS adds padding bits for aligning the size of the data type to a power of 2.

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**



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.

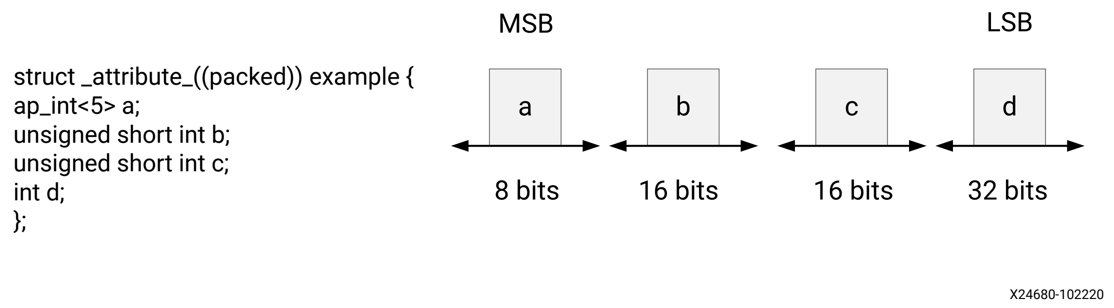
```
struct data_t {
  short varA;
  int varB;
  short varC;
};
```

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**



```
struct _attribute_((packed)) example {
ap_int<5> a;
unsigned short int b;
unsigned short int c;
int d;
};
```

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 bBec = 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	`+=`	`+`	`reduce_add`
Subtraction	`-=`	`-`	*non-associative*
Multiplication	`*=`	`*`	`reduce_mult`
Division	`/=`	`/`	*non-associative*
Remainder	`%=`	`%`	*non-associative*
Bitwise AND	`&=`	`&`	`reduce_and`
Bitwise OR	`|=`	`|`	`reduce_or`
Bitwise XOR	`^=`	`^`	`reduce_xor`
Shift Left	`<<=`	`<<`	*non-associative*
Shift Right	`>>=`	`>>`	*non-associative*
Pre-increment	`++x`	*none*	*unary operator*
Pre-decrement	`--x`	*none*	*unary operator*
Post-increment	`x++`	*none*	*unary operator*
Post-decrement	`x--`	*none*	*unary operator*

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

#define N 85

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

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

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

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

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

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

data_t cpp_FIR(data_t x);
```

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

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

```
#include "cpp_FIR.h"

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


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

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

    result << setw(10) << i;
    result << setw(20) << output;
    result << endl;

 }
 result.close();

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

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

C++ Test Bench for `cpp_FIR`

To apply directives to objects defined in a class:

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

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

Global Variables and Classes

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

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

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

// Use k on line 73 static int k;

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

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

     }
};

// Top-level function with class instantiated
void cpp_class_data (
 acc_t *dataOut,
 coef_t coeff1[PHASES][TAPS],
 coef_t  coeff2[PHASES][TAPS],
 data_t  dataIn[DATA_SAMPLES],
 int   row
) {
 acc_t pcin0 = 0;
 acc_t pcout0, pcout1;
 data_t dout0, dout1;
 int col;
 static acc_t accum=0;
 static int sample_count = 0;
 static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell0;
 static polyd_cell<data_t, acc_t, mult_t, coef_t, CELL_SAMPLES>
polyd_cell1;

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

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

col);

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


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

 }
 sample_count++;
}
```

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

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

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

Templates

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

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 templatized `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 |FF |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 |FF |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.

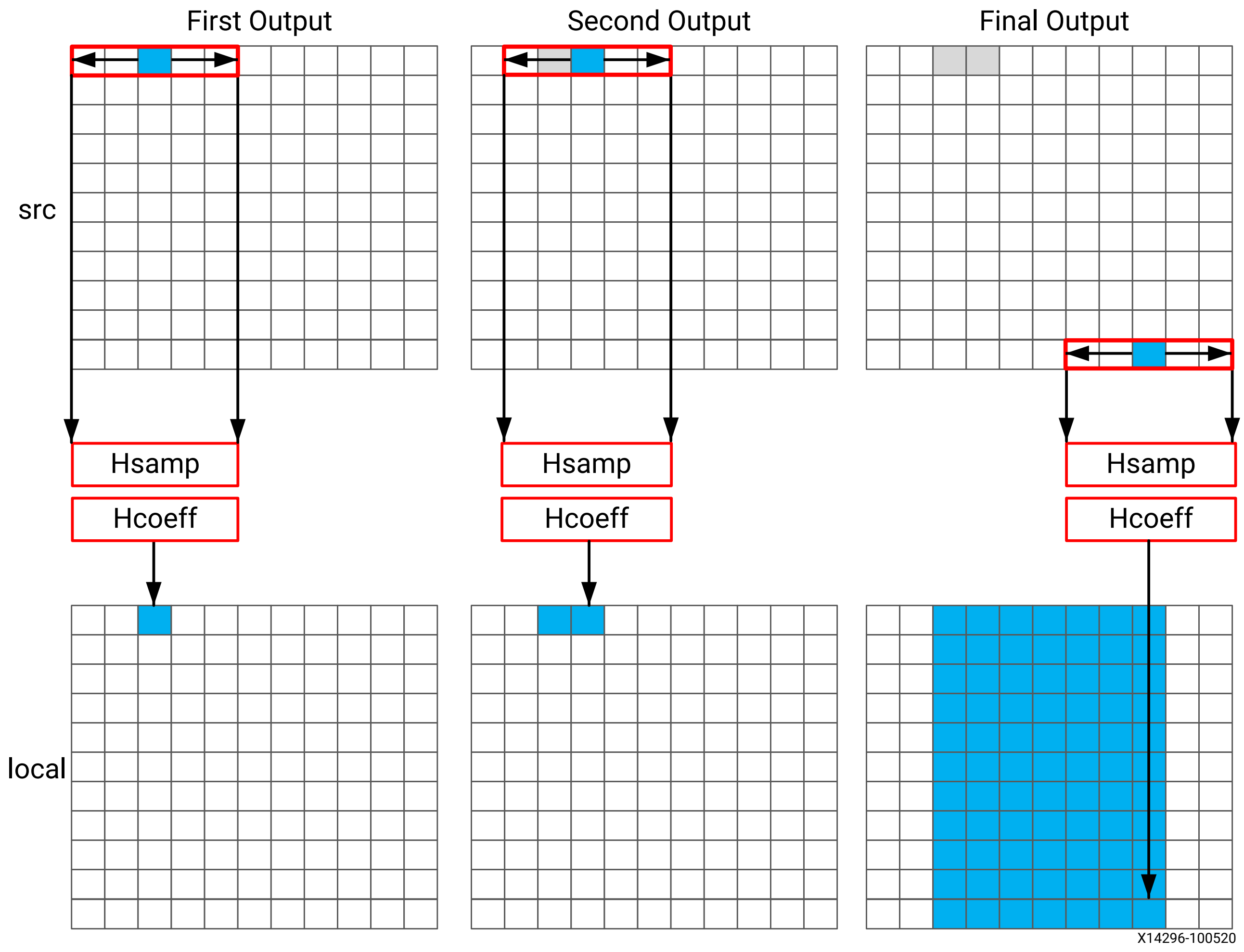
The algorithm structure can be summarized as follows:

```
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++){
 HconvWfor(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 vales. 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.

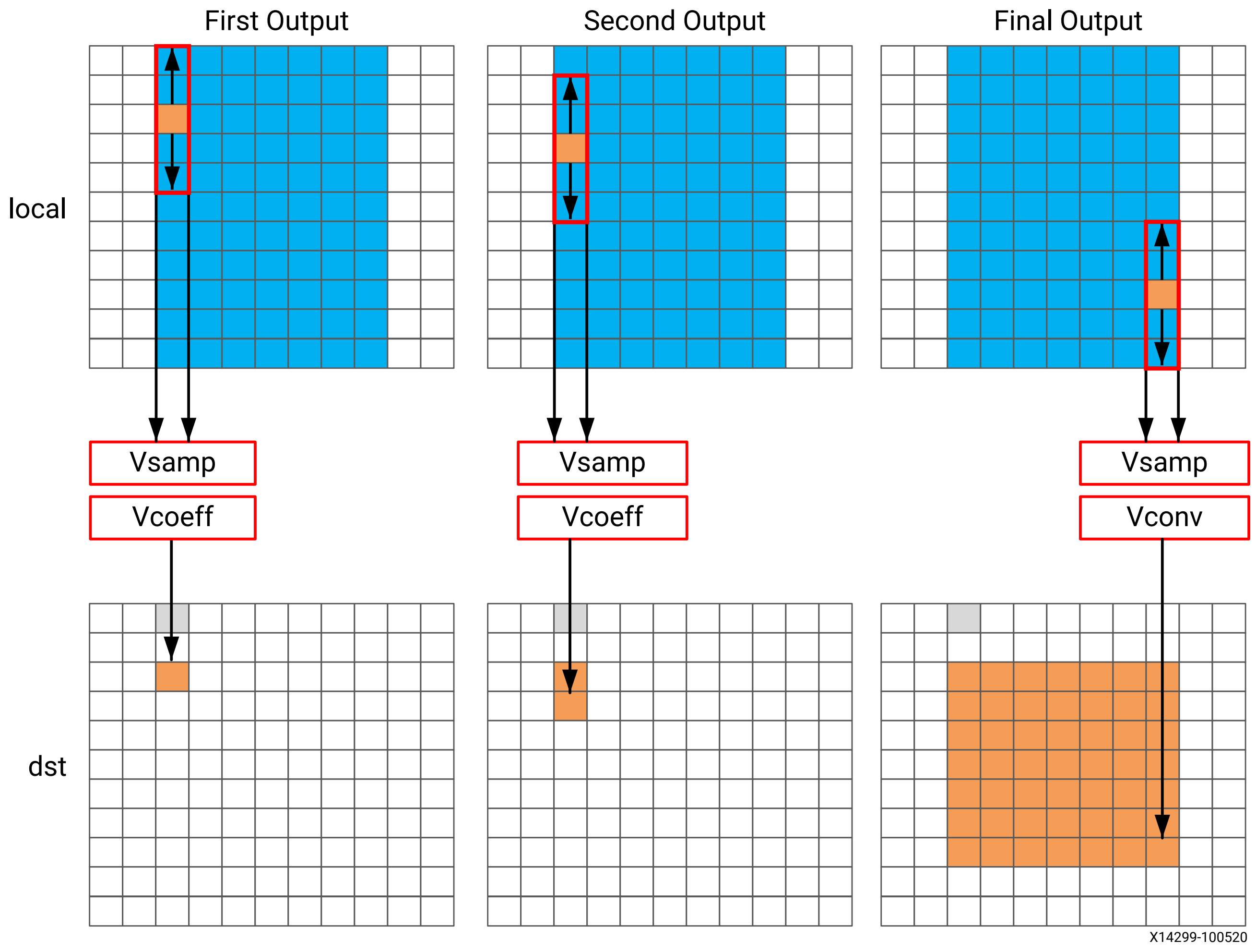
- For the first output, the first K values are read from the input.
- To calculate the second output, the same K-1 values are re-read through the data input port.
- This process of re-reading the data is repeated for the entire image.

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, `Vcoeff` 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 then 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.

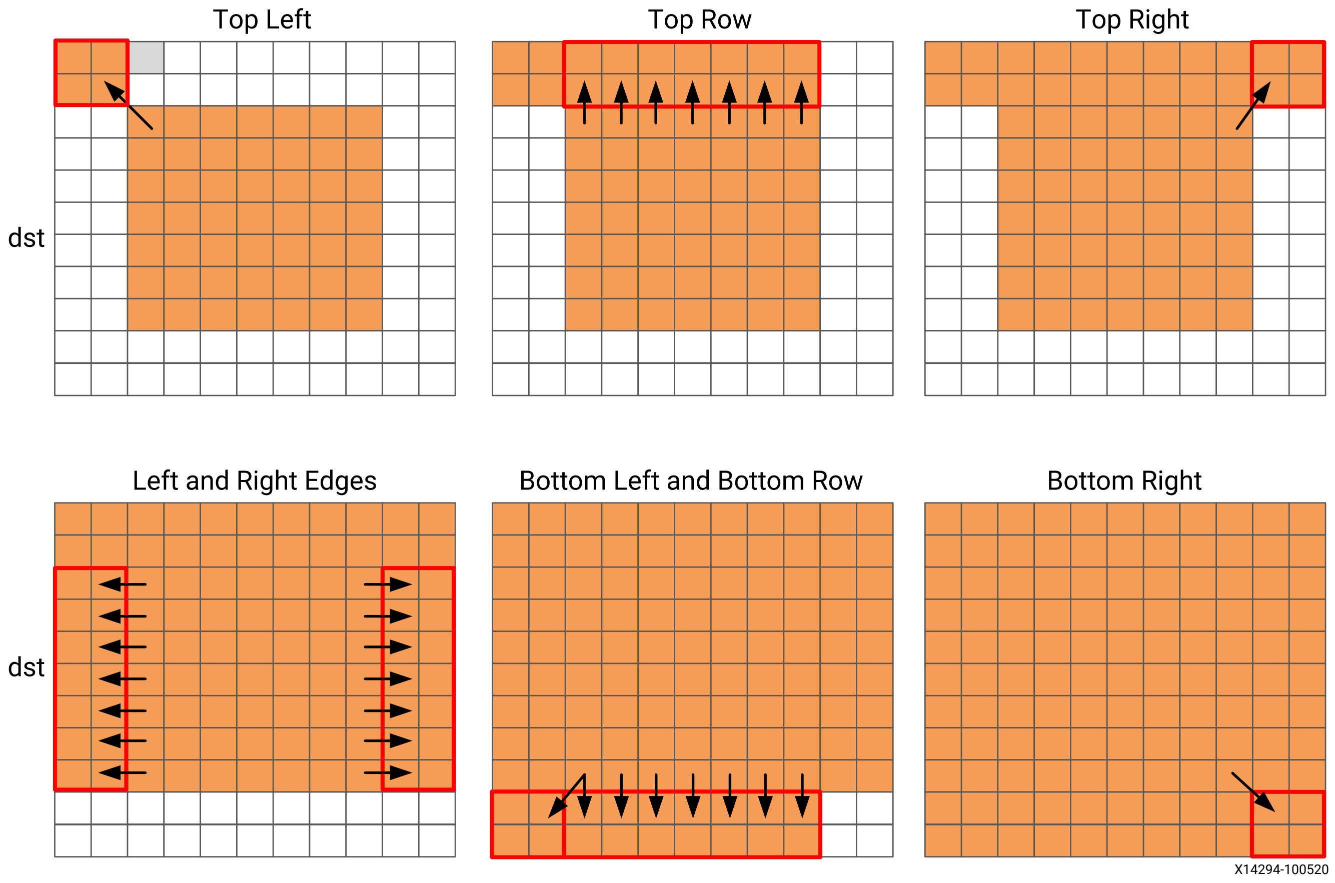
- Many clock cycles are spent to set the values in the output image `dst` to zero. In this case, approximately another 2 million cycles for a 1920*1080 image size.
- There are multiple accesses per pixel to re-read data stored in array `local`.
- There are multiple writes per pixel to the output array/port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

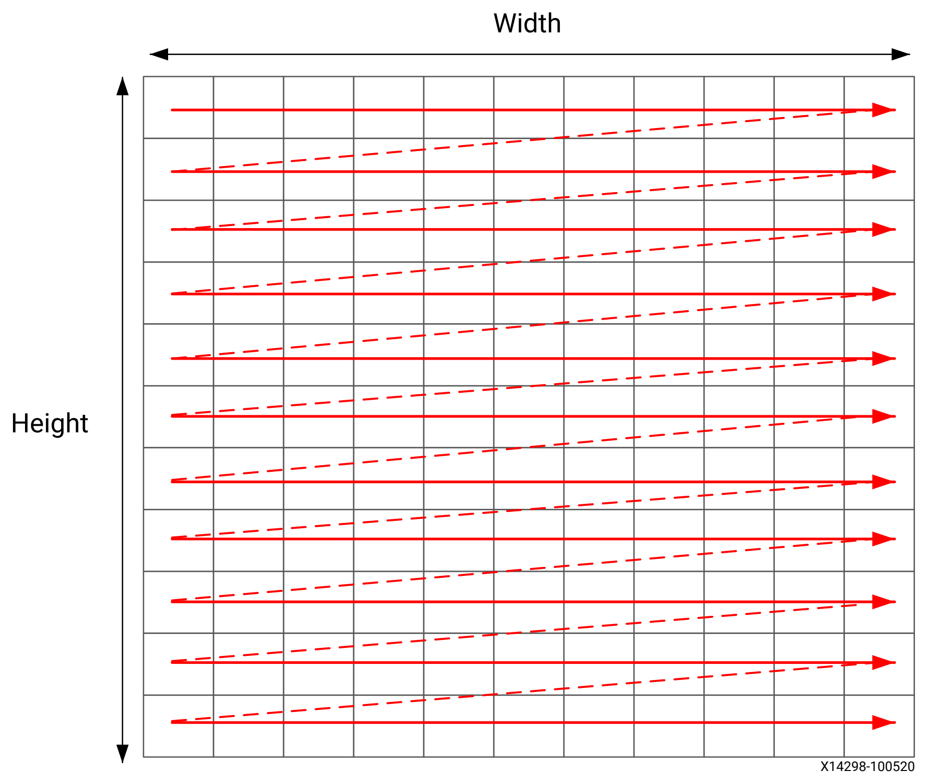
Ensuring the Continuous Flow of Data and Data Reuse

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 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> &dst,
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:

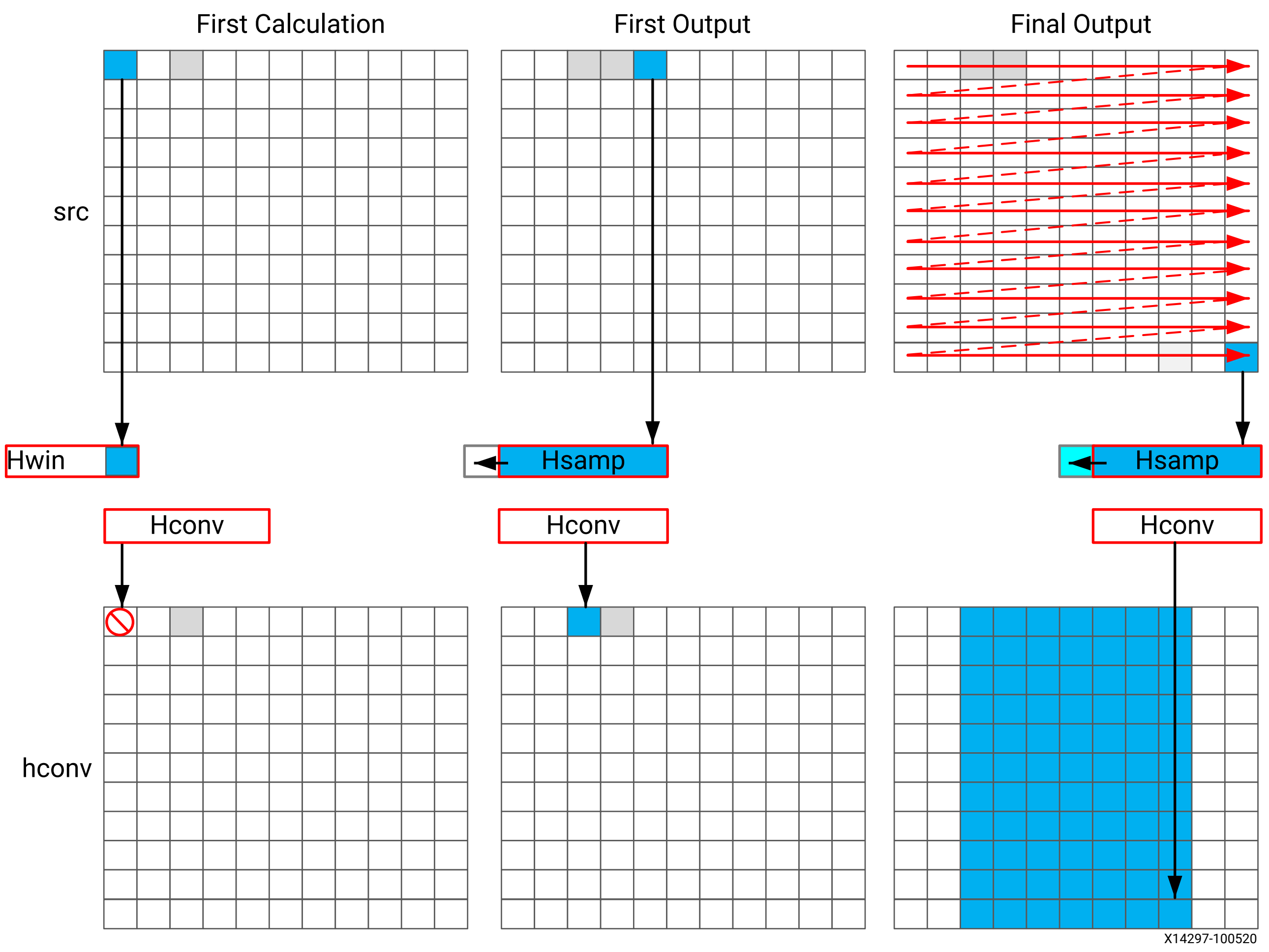
- The input and output data is now modeled as `hls::stream`.
- Instead of a single local array of size HEIGHT*WDITH there are two internal `hls::stream` used to save the output of the horizontal and vertical convolutions.

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 a 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.

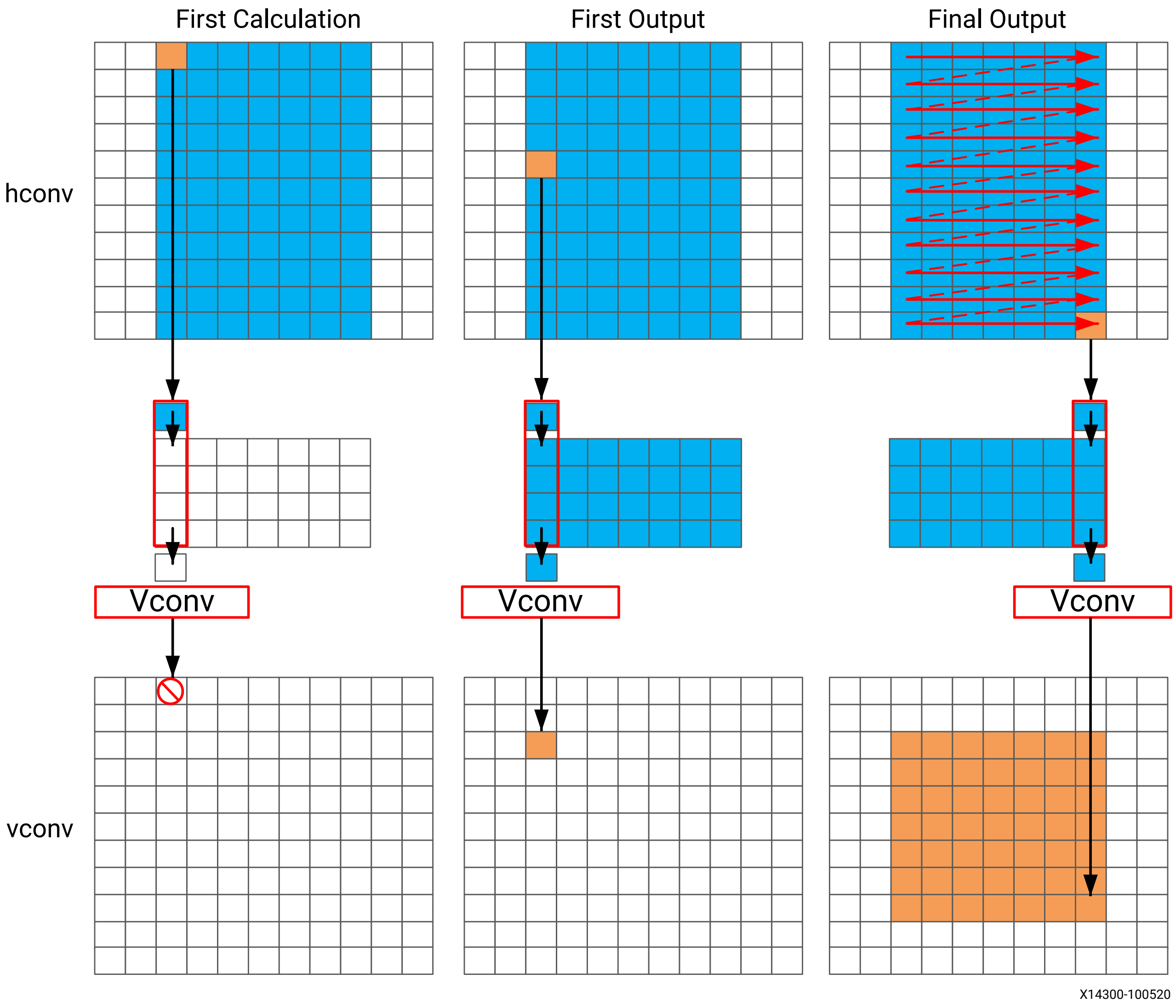
In a CPU architecture, conditional or branch operations are often avoided. When the program needs to branch it loses any instructions stored in the CPU fetch pipeline. In an FPGA architecture, a separate path already exists in the hardware for each conditional branch and there is no performance penalty 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 ensure 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.

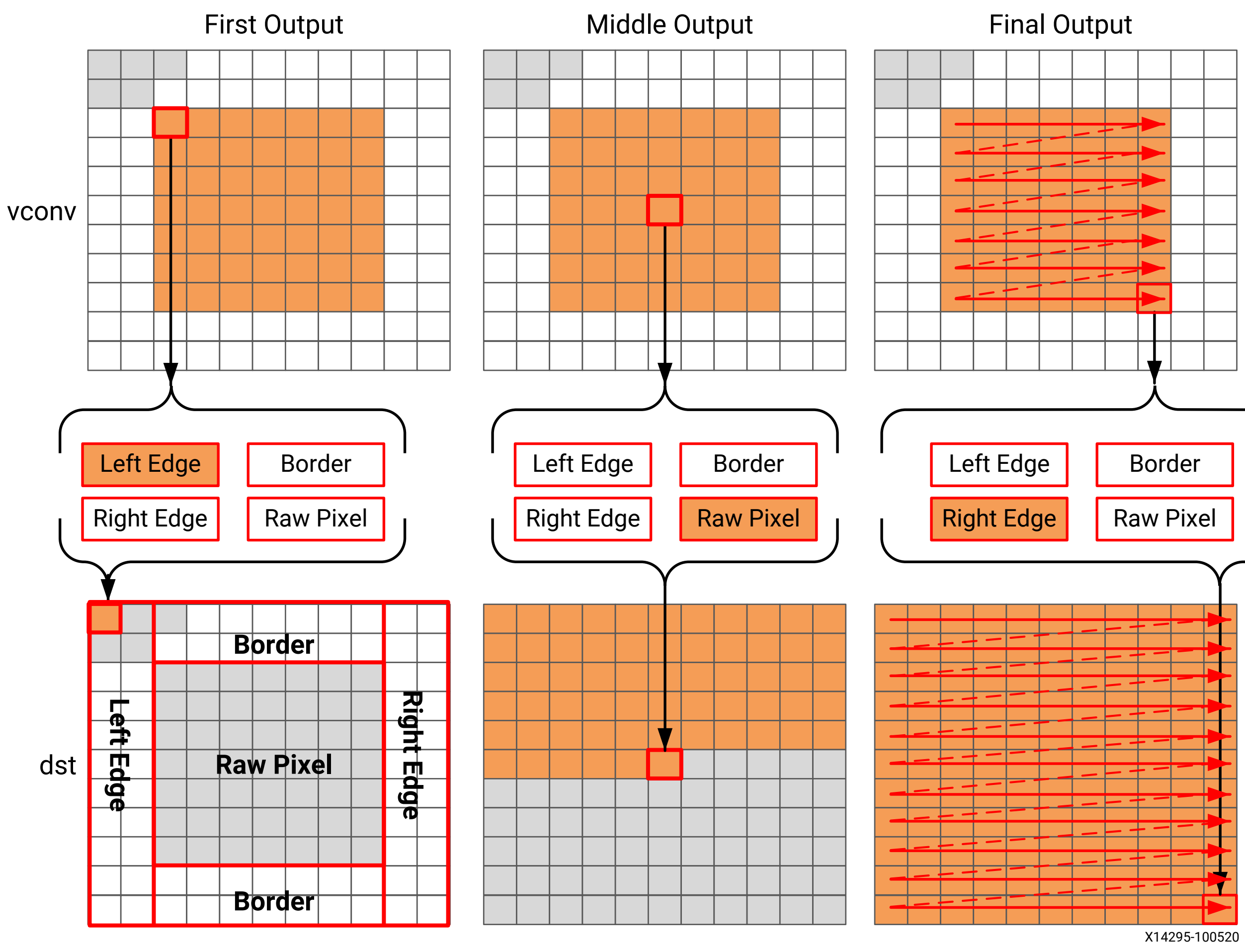
Border Pixels

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 = 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 with 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> &dst,
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.