Programming for Vitis HLS

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 on **Clone Examples** command from the Vitis HLS Welcome screen.

Note: To view the Welcome screen at any time, select **Help** > (and then)**Welcome**.

Unsupported C Constructs

While Vitis HLS supports a wide range of the C language, 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 C 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 constructs must be of a fixed or bounded size.
- The implementation of those constructs must be unambiguous.

For more information, refer to the *Vitis HLS Migration Guide* (UG1391 (https://www.xilinx.com/cgi-bin/docs/rdoc?v=2020.1;d=ug1391-vitis-hls-migration-guide.pdf)).

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 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 simulation or 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 C function. Using such a macro does mean that the C code for simulation and the C code for synthesis are now different.

CAUTION: If the `__SYNTHESIS__` macro is used to change the functionality of the C code, it can result in different results between C simulation and 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 run time. 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 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 simulation, and save the results.
3. Disable the macro `NO_SYNTH` , and execute the 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 simulation and that the identical code is then synthesized. As with restrictions on dynamic memory usage in C, Vitis HLS does not support (for synthesis) C++ objects that are dynamically created or destroyed.

The following code cannot be synthesized because it creates a new function at run time.

```
Class A {
public:
 virtual void bar() {â€¦};
};

void fun(A* a) {
 a->bar();
}
A* a = 0;
if (base)
 a = new A();
else
 a = new B();

foo(a);
```

Pointer Limitations

General Pointer Casting

Vitis HLS does not support general pointer casting, but supports pointer casting between native 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.

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. C++ templates are addressed next.

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.

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 run time as more logic must be kept in memory and analyzed.

> **TIP:** Vitis HLS may 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_cint.h"

int24 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_cint.h"
typedef int12 din_t;
typedef int24 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 Builtin Functions

Vitis HLS supports the following 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 code. Loops are very well supported by synthesis: loops can be pipelined, unrolled, partially unrolled, merged, and flattened.

The optimizations 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.

> **Note:** 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_cint.h"
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 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 (hlspragmas.html#sty1504034367099) or set_directive_loop_tripcount (mko1585343471006.html).
- Use an `assert` macro in the 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 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_cint.h"
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint5 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 run time. 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 run time 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 above, 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 are early as possible to reduce latency. 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 for 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 simulation.

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

```
#include "ap_cint.h"

  int i, acc;
  // Use an arbitrary precision type
  int32  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 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 `-Wl,--stack,10485760` . This can be applied in Vitis™ HLS by going to **Project Settings** > (and then)**Simulation** > (and then)**Linker flags**, or it can also be provided as options to the Tcl commands:

```
csim_design -ldflags {-Wl,--stack,10485760}
cosim_design -ldflags {-Wl,--stack,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 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_cint.h"

  int i, acc;
#ifdef __SYNTHESIS__
  // Use an arbitrary precision type & array for synthesis
  int32  la0[10000000], la1[10000000];
#else
  // Use an arbitrary precision type & dynamic memory for simulation
 int32 *la0 = malloc(10000000  * sizeof(int32));
 int32 *la1 = malloc(10000000  * sizeof(int32));
#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 simulation or 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 SRL. Arrays sized greater than 1024 will be synthesized into block RAM, LUTRAM, UltraRAM depending on the optimization settings.

Like loops, arrays are an intuitive coding construct and so they are often found in 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. Array initialization, if not performed carefully, can result in undesirably long reset and initialization in the RTL.
- 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. For example, sized arrays are supported, 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 (and 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 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.

Smaller arrays can also be combined into a single larger array, resulting in a single interface. While this might map better to an off-chip block RAM, it might also introduce a performance bottleneck. These trade-offs can be made using Vitis HLS optimization directives and do not impact coding of the function.

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 determines whether the accesses are sequential.

Table 1. 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 ap_fifo port=d_i
#pragma HLS INTERFACE 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 `€œ//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 streaming interface (instead of 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

Note: 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 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

The data types used in a 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 run time is impacted because it typically requires multiple accesses to read and write those values.

Similarly, when the 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 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 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 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-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 data types can result in inefficient hardware implementation. For example the basic multiplication unit in an FPGA is the DSP module. This provides a multiplier which is 18*18-bit. If a 17-bit multiplication is required, you should not be forced to implement this with a 32-bit C data type: this would require three DSP48 macros to implement a multiplier when only one is required.

Arbitrary precision (AP) data types allow your code to use variables with smaller bit-widths, and for the 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 (vitishlslibrariesreference.html#ogi1585574074269).

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 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

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 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 types but in some width which is smaller, but still greater than the next smallest standard 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 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 2. Arbitrary Precision Data Types

Language	Integer Data Type	Required Header
C++	ap_[u]int<W> (1024 bits) Can be extended to 32K bits wide.	#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 - only with Vitis HLS.

Arbitrary Precision Types with C++

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 32768 before inclusion of the `ap_int.h` header file.

CAUTION: Setting the value of `AP_INT_MAX_W` too high can cause slow software compile and run times.

CAUTION: 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] =
```

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. In this example the Vitis HLS `ap_fixed` type is used to define an 18-bit variable with 6 bits representing the numbers above the binary point and 12-bits representing the value below the decimal point. The variable is specified as signed, the quantization mode is set to round to plus infinity. Since 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 3. Fixed-Point Identifier Summary

Identifier	**Description**		
W	Word length in bits		
I	The number of bits used to represent the integer value (the number of bits above the binary point)		
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) value which can be stored in the result variable.		
		ap_fixed Types	Description
		AP_SAT	Saturation
		AP_SAT_ZERO	Saturation to zero
		AP_SAT_SYM	Symmetrical saturation
		AP_WRAP	Wrap around (default)
		AP_WRAP_SM	Sign magnitude wrap around
N	This defines the number of saturation bits in the overflow wrap modes.		

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 32768 before inclusion of the `ap_int.h` header file.

CAUTION: Setting the value of `AP_INT_MAX_W` too High may cause slow software compile and run times.

CAUTION: ROM synthesis can be slow when: `static APFixed_2_2 CAcode_sat[32][CACODE_LEN] =` . Changing `APFixed` to `int` results in a faster synthesis: `static int CAcode_sat[32][CACODE_LEN] =`

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_fixed.h"

ap_fixed<4096> very_wide_var;
```

Arbitrary precision data types are highly recommend when using Vitis HLS. As shown in the earlier example, they typically have a significant positive benefit on the quality of the hardware implementation.

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 compliance.

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

Note: When using floating-point data types, Xilinx highly recommends that you review *Floating-Point Design with Vitis HLS* (XAPP599 (https://www.xilinx.com/cgi-bin/docs/ndoc?t=application_notes;d=xapp599-floating-point-vivado-hls.pdf))

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.

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

Arbitrary Precision Data Types

Vitis HLS provides arbitrary precision data types as described in Arbitrary Precision (AP) Data Types.

Composite Data Types

Vitis HLS supports composite data types for synthesis:

- Structs
- Enumerated Types
- Unions

Structs

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

IMPORTANT: Structs used as arguments to the top-level function are aggregated by default, and can not be disaggregated. To disaggregate a struct on the interface, you must manually code it as separate elements.

Alternatively, the AGGREGATE pragma or directive can be used for collecting 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 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: Care should be taken when using the AGGREGATE optimization 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. Vitis HLS can create this RTL design, however it is very unlikely that logic synthesis will be able to route this during the FPGA 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.

Struct Padding and Alignment

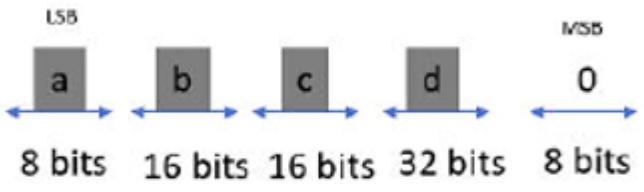
Structs in Vitis HLS can have different types of padding and alignment depending on the use of `__attributes__` or `#pragmas`. These features are described below.

Disaggregate

By default, structs in the code as internal variables are disaggregated. Structs defined on the kernel interface are not. Disaggregated structs are broken up into individual elements as described in set_directive_disaggregate (https://www.xilinx.com/cgi-bin/docs/rdoc?t=vitis+doc;v=2020.1;d=hlsoptimizationdirectives.html;a=hhv1584808444510), or pragma HLS disaggregate (https://www.xilinx.com/cgi-bin/docs/rdoc?t=vitis+doc;v=2020.1;d=hlspragmas.html;a=lbk1584844390084). You do not need to apply the DISAGGREGATE pragma or directive, as this is the default behavior for the struct.

Figure 1: Disaggregated Struct

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



Aggregate

This is the default behavior for structs on the interface, as discussed in Interface Synthesis and Structs (https://www.xilinx.com/html_docs/xilinx2020_1/vitis_doc/programmingvitishls.html#kdg1585178577707). Vitis HLS joins the elements of the struct, aggregating the struct into a single data unit. This default is done in accordance with pragma HLS aggregate (https://www.xilinx.com/cgi-bin/docs/rdoc?t=vitis+doc;v=2020.1;d=hlspragmas.html;a=uhk1586265569642), although you do not need to specify the pragma as this is the default for structs on the interface. The aggregate process may also involve data padding for elements of the struct, to align the byte structures on a default 4-byte alignment.

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

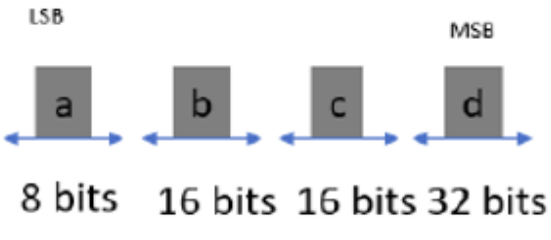
Aligned

By default Vitis HLS will align struct on a 4-byte alignment, padding elements of the struct to align it to a 32-bit width. However, you can use the `__attribute__((aligned(X)))` to add padding to elements of the struct, to align it on "X" byte boundaries. In the figure below, the struct is aligned on a 2-byte boundary

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

Figure 2: Aligned Struct Implementation

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



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

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

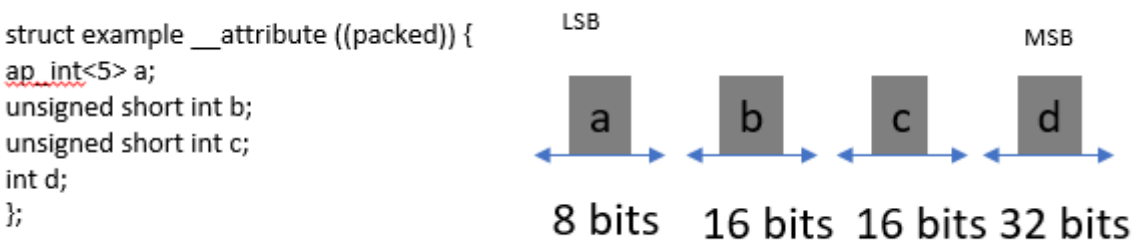
However, if you rewrite the struct as follows, there will be no need for padding, and the total size of the struct will be 64-bits.

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

Packed

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

Figure 3: Packed Struct Implementation




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

Data Layout for Arbitrary Precision Types (ap_int library)

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

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

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

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

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;
```

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 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 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) & (0x7fffffffffffffffLL);
}
```

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 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 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: Volatile accesses are forbidden from being optimized by design. This means:

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

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 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 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 code and are well-supported for synthesis. When using pointers, be careful 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 types, as shown.

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 run time.

```
#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 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
```

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. The code in the Interface with Pointer Arithmetic example can be synthesized only with an `ap_bus` interface. This interface supplies an address with which to index the data when the data is accessed (read or write).

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 in conjunction 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 Pointer Interfaces: Streaming Data

Designs that use pointers in the argument list of the top-level function 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.

- You must use the `volatile` qualifier on any function argument accessed multiple times.
- On the top-level function, any such argument must have the number of accesses on the port interface specified if you are verifying the RTL using co-simulation within Vitis HLS.
- Be sure to validate the C before synthesis to confirm the intent and that the C model is correct.

If modeling the design requires that an function argument be accessed multiple times, Xilinx recommends that you model the design using streams. Use streams to ensure that you do not encounter the issues discussed in this section.

Table 4. Example Design Scenarios

Example Design	**Shows**
`pointer_stream_bad`	Why the `volatile` qualifier is required when accessing pointers multiple times within the same function.
`pointer_stream_better`	Why any design with such pointers on the top-level interface should be verified with a C test bench to ensure that the intended behavior is correctly modeled.

In the following code example, input pointer `d_i` is read from four times and output `d_o` is written to twice, with the intent that the accesses are implemented by FIFO interfaces (streaming data into and out of the final RTL implementation).

```
#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;
}
```

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;
}
```

Understanding Volatile Data

The code in the Multi-Access Pointer Interface example 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 blocks supply new data each time a read is performed on RTL port `d_i` .
- Downstream consumer blocks accept new data each time there is a write to RTL port `d_o` .

When this code is compiled by standard 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 simulation of Multi-Access Pointer Interfaces: Streaming Data 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. See the following code example.

  The `volatile` qualifier tells the C compiler (and Vitis HLS) to make no assumptions about the pointer accesses. That is, the data is volatile and might change.

  TIP: Do not optimize pointer accesses.

```
#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;
}
```

  The example above simulates the same as Multi-Access Pointer Interfaces: Streaming Data, but 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, this coding style (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.

Note: To see the intermediate accesses, enable `cosim_design` to create a trace file during RTL simulation and view the trace file in the appropriate viewer.

The Multi-Access Volatile Pointer Interface example above 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 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);
 }
```

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.

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 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) {
```

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.

Constructors, Destructors, and Virtual Functions

Class constructors and destructors are included and synthesized whenever a class object is declared.

Vitis HLS supports virtual functions (including abstract functions) for synthesis, provided that it can statically determine the function during elaboration. Vitis HLS does not support virtual functions for synthesis in the following cases:

- Virtual functions can be defined in a multilayer inheritance class hierarchy but only with a single inheritance.
- Dynamic polymorphism is only supported if the pointer object can be determined at compile time. For example, such pointers cannot be used in an if-else or loop constructs.
- An STL container cannot contain the pointer of an object and call the polymorphism function. For example:

```
vector<base *> base_ptrs(10);

//Push_back some base ptrs to vector.
for (int i = 0; i < base_ptrs.size(); ++i) {
 //Static elaboration cannot resolve base_ptrs[i] to actual data type.
 base_ptrs[i]->virtual_function();
}
```

- Vitis HLS does not support cases in which the base object pointer is a global variable. For example:

```
Base *base_ptr;

void func()
{
 ...
 base_prt->virtual_function();
 ...
}
```

- The base object pointer cannot be a member variable in a class definition. For example:

```
// Static elaboration cannot bind base object pointer with correct data type.
class A
{
 ...
 Base *base_ptr;
 void set_base(Base *base_ptr);
 void some_func();
 ...
};

void A::set_base(Base *ptr)
{
 this.base_ptr = ptr;
}

void A::some_func()
{
 â€¦.
 base_ptr->virtual_function();
 â€¦.
}
```

- If the base object pointer or reference is in the function parameter list of constructor, Vitis HLS does not convert it. The ISO C++ standard has depicted this in section12.7: sometimes the behavior is undefined.

```
class A {
 A(Base *b) {
 b-> virtual _ function ();
 }
};
```

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;
```

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.

```
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 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

The assert macro in 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_cint.h
#define N 32

typedef int8 din_t;
typedef int13 dout_t;
typedef uint8 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 simulation will fail. This also highlights why it is important to simulate the 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 tripcount 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 Tripcount 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 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 device 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 device - 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.

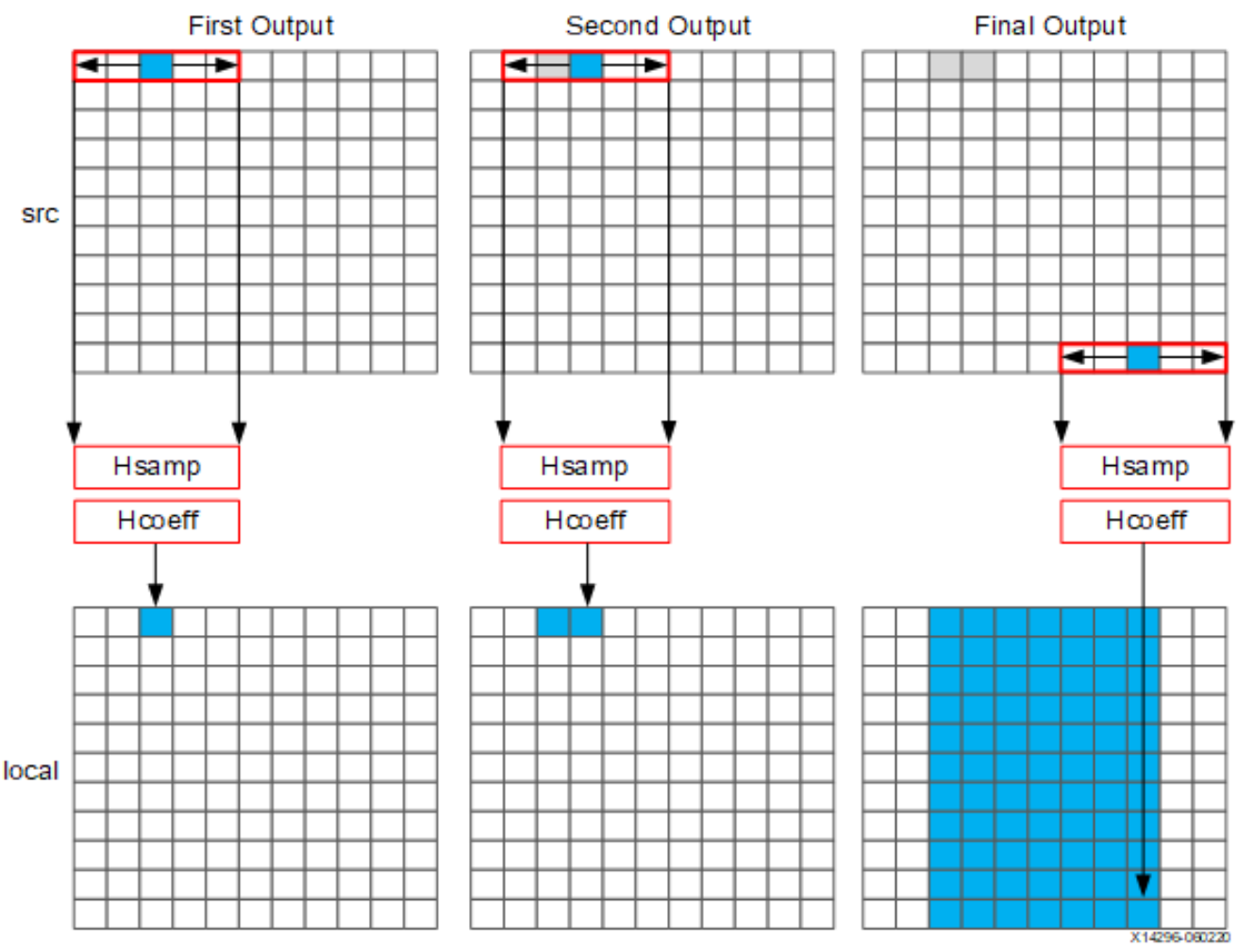
The algorithm structure can be summarized as follows:

```
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 4: 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, since 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 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 simulation or C RTL co-simulation.

The code is straight forward and intuitive. There are already however some issues with this 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 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 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` . Since 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.

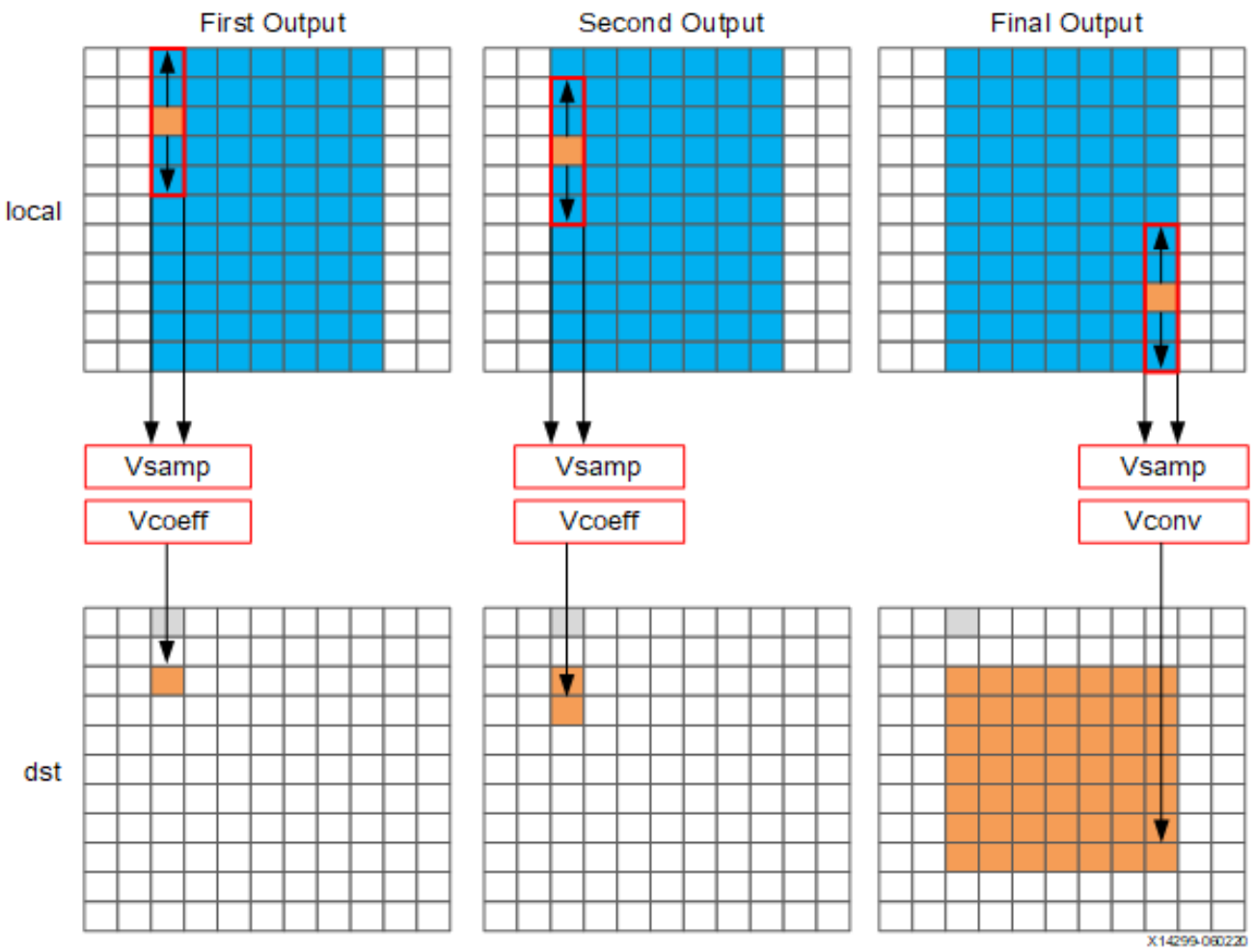
- For the first output, the first K values are read from the input.
- To calculate the second output, the same K-1 values are re-read through the data input port.
- This process of re-reading the data is repeated for the entire image.

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, since 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 5: 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.

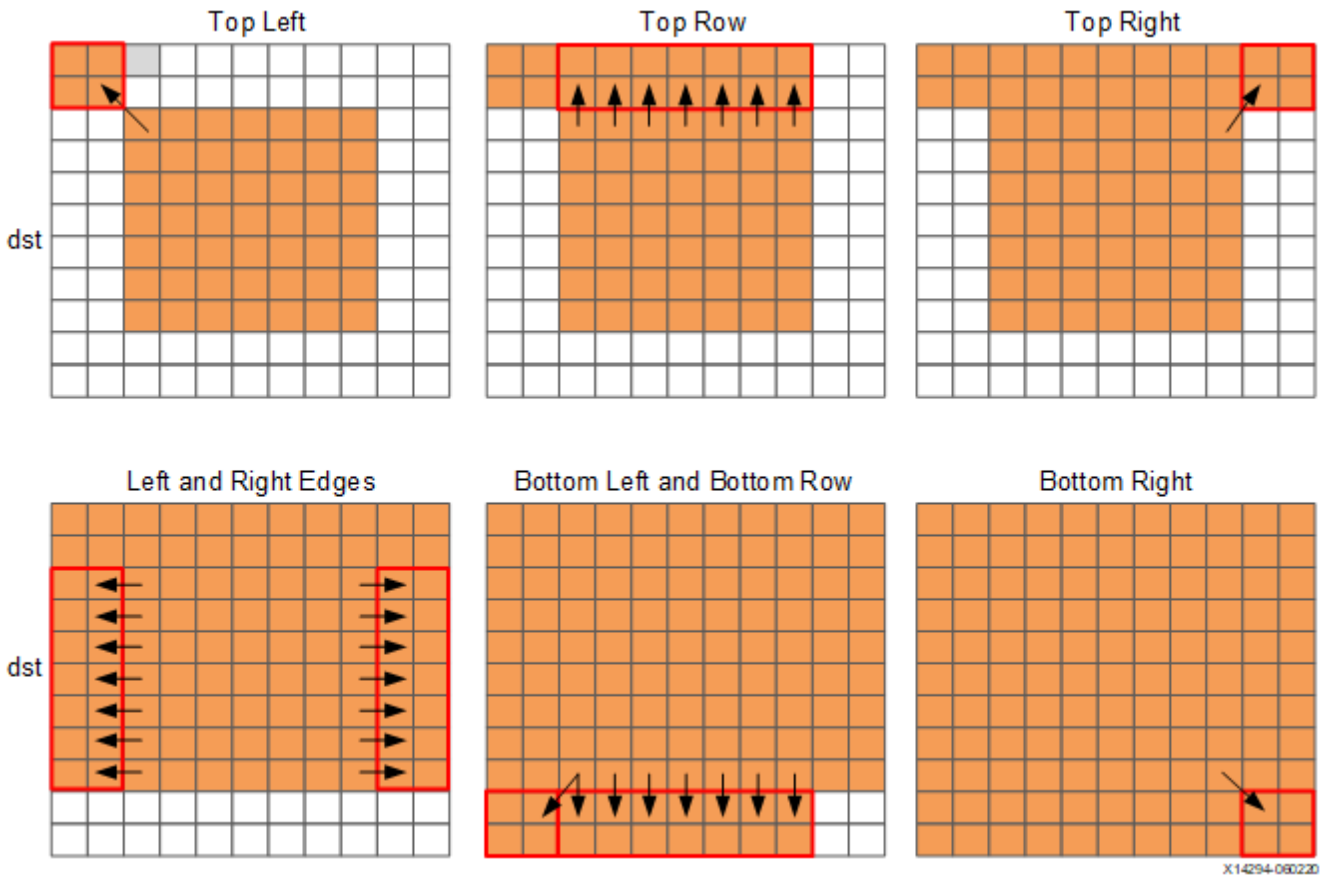
- Many clock cycles are spent to set the values in the output image `dst` to zero. In this case, approximately another 2 million cycles for a 1920*1080 image size.
- There are multiple accesses per pixel to re-read data stored in array `local` .
- There are multiple writes per pixel to the output array/port `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 6: 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++){
```

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.

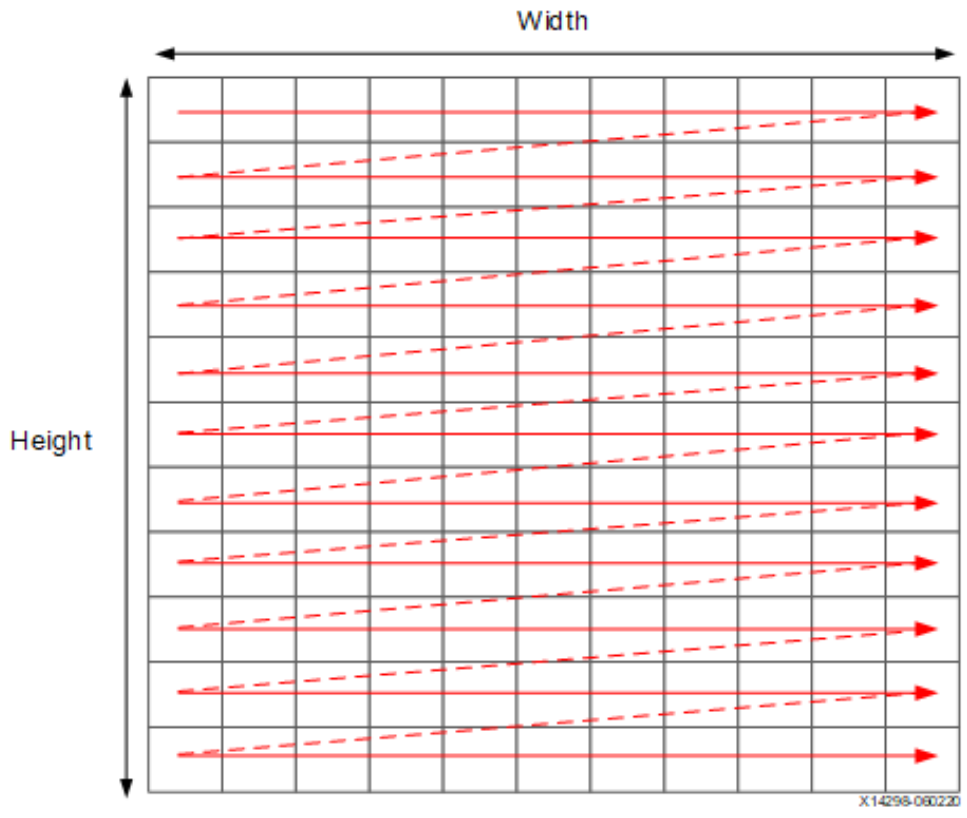
Ensuring the Continuous Flow of Data and Data Reuse

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 7: 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 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 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 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:

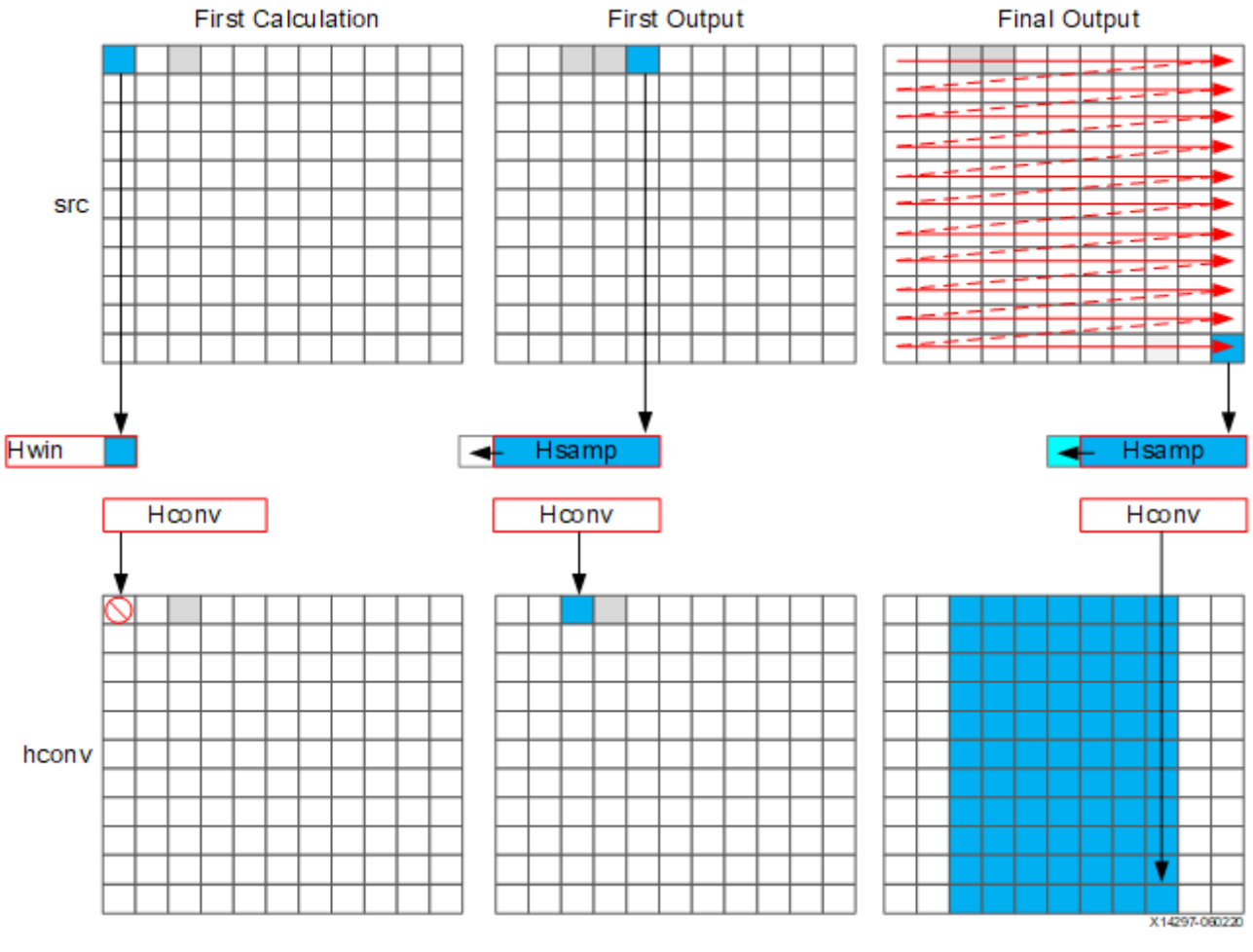
- The input and output data is now modeled as `hls::stream` .
- Instead of a single local array of size HEIGHT*WDITH there are two internal `hls::stream` used to save the output of the horizontal and vertical convolutions.

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 8: 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 is 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 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.

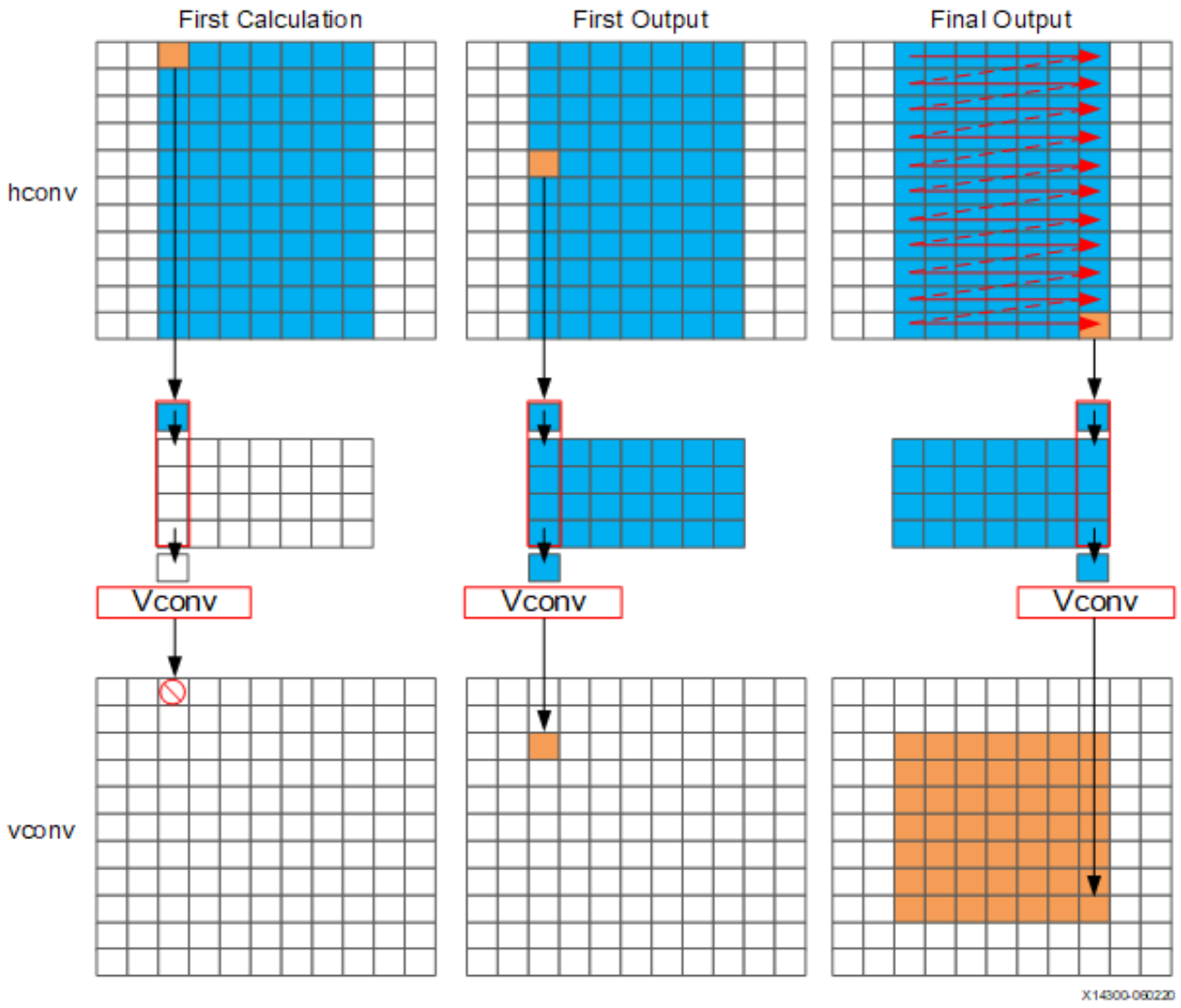
In a CPU architecture, conditional or branch operations are often avoided. When the program needs to branch it loses any instructions stored in the CPU fetch pipeline. In an FPGA architecture, a separate path already exists in the hardware for each conditional branch and there is no performance penalty 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 9: 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 inter 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.

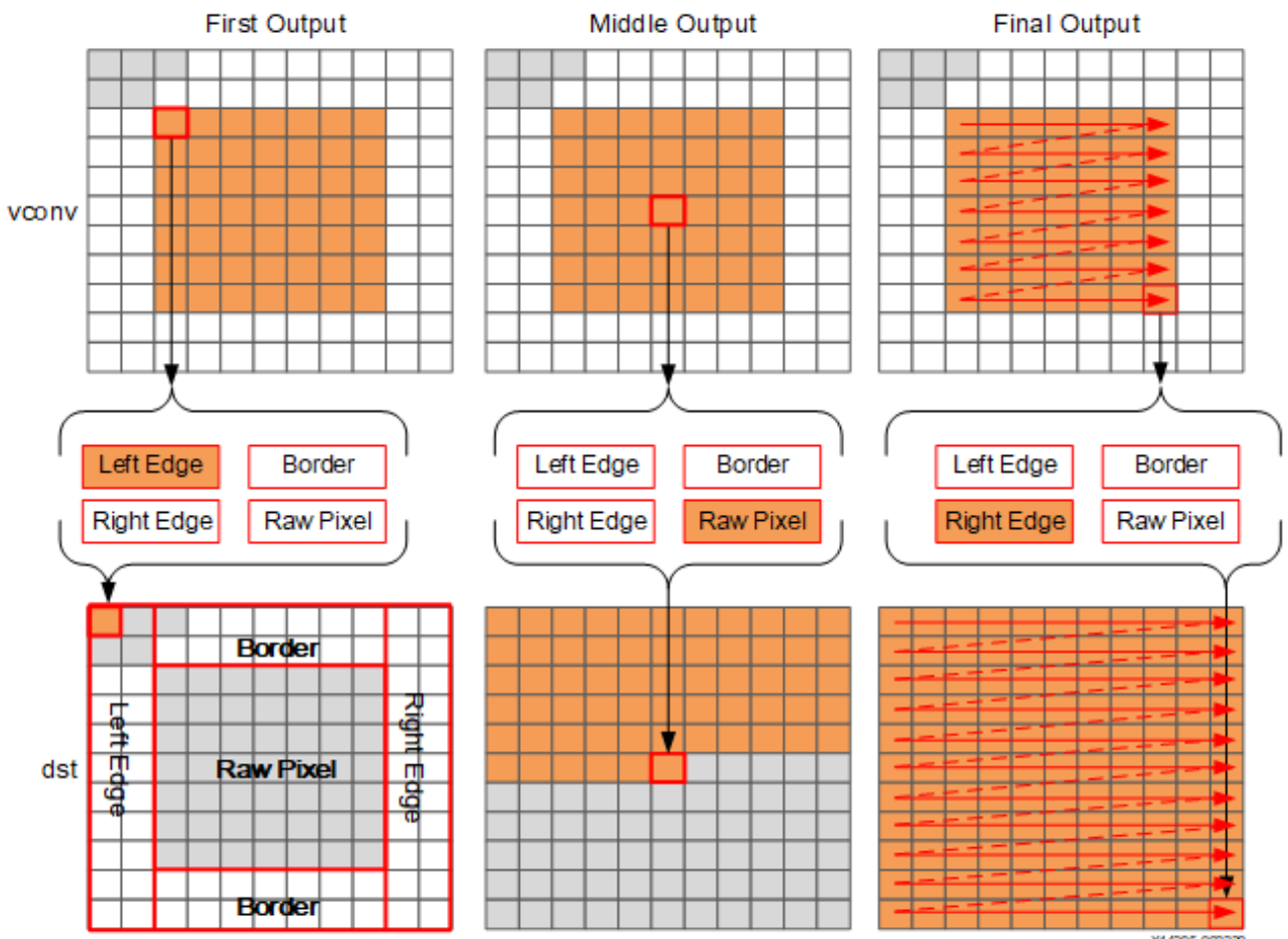
Border Pixels

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 4 possible pixel types.
- The sample is then written to the output stream.

Figure 10: 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 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 inter false
   VConv:for(int i = 0; i < K; i++) {
 }
}
```

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 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.

Managing Interface Synthesis

In C based design, all input and output operations are performed, in zero time, through formal function arguments. In an RTL design these same input and output operations must be performed through a port in the design interface, and typically operates using a specific I/O (input-output) protocol. Vitis HLS defines this port interface automatically, using industry standards to specify the I/O protocol used.

When the top-level function is synthesized, the arguments to the function are synthesized into RTL ports. This process is called *interface synthesis*.

Vitis HLS performs interface synthesis by default, selecting the port interface and I/O protocols based on the target flow, and the data type and direction of the function argument. The target flow is specified through the GUI as described in Creating a New Vitis HLS Project (creatingnewvitishlsproject.html#thg1583443745171), or can be defined by specifying the target in the open_solution (ode1585337030434.html) command.

When specifying `open_solution -flow_target vitis`, or enabling the **Vitis Kernel Flow** in the GUI, Vitis HLS implements interface ports using the AXI standard as described in Port-Level I/O: AXI4 Interface Protocol, and Using AXI4 Interfaces.

IMPORTANT: You can override the default interface specification of the tool by manually specifying the INTERFACE pragma or directive for a function argument. This lets you customize the features and performance of your hardware implementation, but may not meet the kernel requirements for use in the Vitis application acceleration development flow.

Interface Synthesis Overview

In the following code, the `sum_io` function provides an overview of interface synthesis.

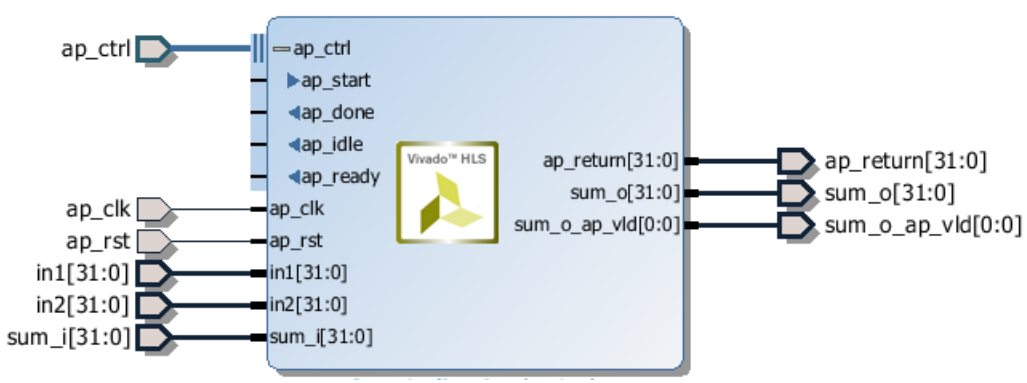
```
#include "sum_io.h"

dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {

 dout_t temp;

 *sum = in1 + in2 + *sum;
 temp = in1 + in2;

 return  temp;
}
```

The `sum_io` example includes:

- Two pass-by-value inputs: `in1` and `in2`.
- A pointer: `sum` that is both read from and written to.
- A function `return` assigned the value of `temp`.

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

Figure 11: RTL Ports After Default Interface Synthesis



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

- Clock and Reset ports: `ap_clk` and `ap_rst`.
- Block-Level interface protocol: These are shown on the expanded `ap_ctrl` interface in the preceding figure: `ap_start`, `ap_done`, `ap_ready`, and `ap_idle`.
- Port Level interface protocols: These are created for each argument in the top-level function and the function return (if the function returns a value). In the example above, these ports include: `in1`, `in2`, `sum_i`, `sum_o`, `sum_o_ap_vld`, and `ap_return`.

Clock and Reset Ports

If the design takes more than 1 cycle to complete operation, a chip-enable port can optionally be added to the entire block using **Solution** > (and then)**Solution Settings** > (and then)**General** and the `config_interface` (lls1585342611108.html) command.

The operation of the reset is controlled by the `config_rtl` (ufi1585342763037.html) configuration.

Block and Port Interface Protocols

Block-level interface protocols provide a mechanism for controlling the operation of the RTL module from other modules, and from software applications. Port-level interface protocols provide the same control over individual ports on the RTL module.

By default, a block-level interface protocol is added to the design to control the block. The ports of a block-level interface control when the block can start processing data (`ap_start`), indicate when it is ready to accept new inputs (`ap_ready`), and indicate if the design is idle (`ap_idle`) or has completed operation (`ap_done`). These are discussed in detail in Block-Level I/O Protocols.

Port-level I/O protocols are control signals assigned to the data ports of the RTL module. The I/O protocol created depends on the type of C argument and on the default. After the block-level protocol has been used to start the operation of the block, the port-level IO protocols are used to sequence data into and out of the block. Details are provided in Port-Level I/O Protocols.

Interface Synthesis I/O Protocols

The type of interfaces that are created by interface synthesis depends on the type of C argument, the target flow, the default interface mode, and the INTERFACE optimization directive. The following figure shows the interface protocol mode you can specify on each type of C argument. This figure uses the following abbreviations:

- D: Default interface mode for each type.

  Note: If you specify an illegal interface, Vitis HLS issues a message and implements the default interface mode.

- I: Input arguments, which are only read.
- O: Output arguments, which are only written to.
- I/O: Input/Output arguments, which are both read and written.

In the Vitis kernel flow, `open_solution -flow_target vitis` , if there are no existing INTERFACE pragmas in the code, then following interfaces will be automatically applied.

Table 5. Argument Types

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

TIP: The `s_axi_lite` interface is the default interface for scalar inputs, as well as the I/O port control interface for all ports except streaming.

The following sections provides an overview of each interface mode, and full details on the interface protocols, including waveform diagrams.

Block-Level I/O Protocols

Vitis HLS uses the interface types `ap_ctrl_none` , `ap_ctrl_hs` , and `ap_ctrl_chain` to specify whether the RTL is implemented with block-level handshake signals. Block-level handshake signals specify the following:

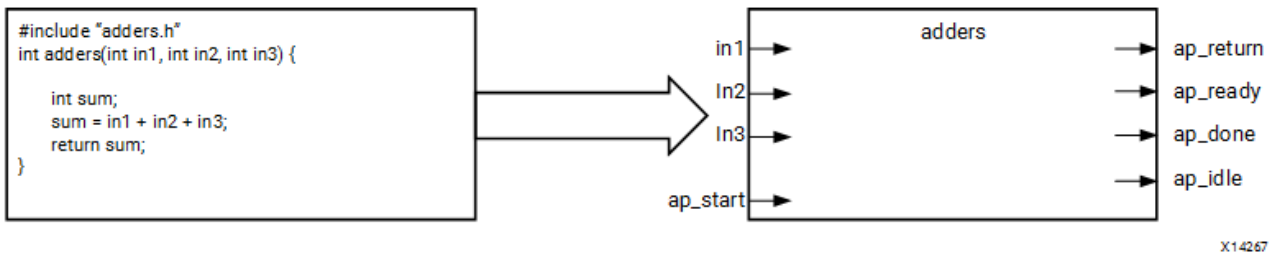
- When the design can start to perform the operation
- When the operation ends
- When the design is idle and ready for new inputs

You can specify these block-level I/O protocols on the function or the function return. If the C code does not return a value, you can still specify the block-level I/O protocol on the function return. If the C code uses a function return, Vitis HLS creates an output port `ap_return` for the return value.

TIP: If the function return is specified as an AXI4-Lite interface (`s_axilite`) all the ports in the block-level interface are grouped into the AXI4-Lite interface, called `control` by default. This is a common practice when another device, such as a CPU, is used to configure and control when the block starts and stops operation.

The `ap_ctrl_hs` block-level I/O protocol is the default. The following figure shows the resulting RTL ports and behavior when Vitis HLS implements `ap_ctrl_hs` on a function. In this example, the function returns a value using the `return` statement, and Vitis HLS creates the `ap_return` output port in the RTL design. If a function `return` statement is not included in the C code, this port is not created.

Figure 12: Example ap_ctrl_hs Interface



The `ap_ctrl_chain` interface mode is similar to `ap_ctrl_hs` but provides an additional input signal `ap_continue` to apply back pressure. Xilinx recommends using the `ap_ctrl_chain` block-level I/O protocol when chaining Vitis HLS blocks together.

ap_ctrl_none

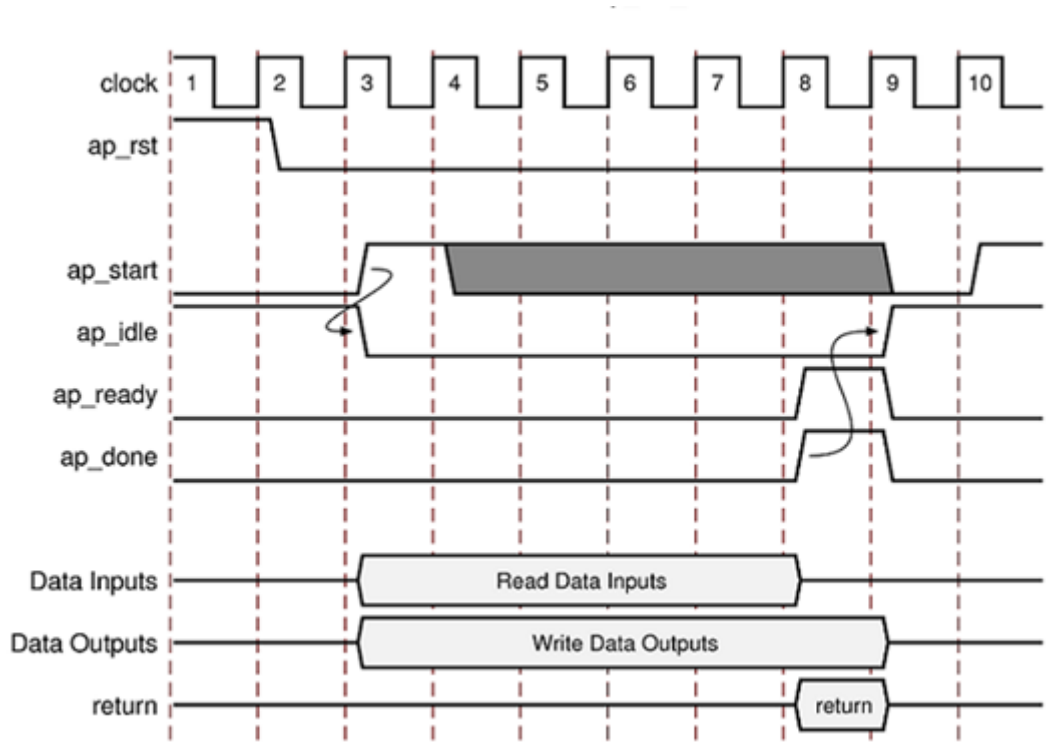
If you specify the `ap_ctrl_none` block-level I/O protocol, the handshake signal ports (`ap_start` , `ap_idle` , `ap_ready` , and `ap_done`) shown in the figure above are not created. You can use this protocol to create a block without control signals, as used in free-running kernels.

IMPORTANT: If you use the `ap_none` block-level I/O protocol on your design, you must meet at least one of the conditions for C/RTL co-simulation as described in Interface Synthesis Requirements (cosimulationinvitishls.html#iox1539734229572) to verify the RTL design.

ap_ctrl_hs

The following figure shows the behavior of the block-level handshake signals created by the `ap_ctrl_hs` I/O protocol for a non-pipelined design.

Figure 13: Behavior of ap_ctrl_hs Interface



After reset, the following occurs:

1. The block waits for `ap_start` to go High before it begins operation.
2. Output `ap_idle` goes Low immediately to indicate the design is no longer idle.
3. The `ap_start` signal must remain High until `ap_ready` goes High. Once `ap_ready` goes High:
   - If `ap_start` remains High the design will start the next transaction.
   - If `ap_start` is taken Low, the design will complete the current transaction and halt operation.
4. Data can be read on the input ports.
5. Data can be written to the output ports.

   Note: The input and output ports can also specify a port-level I/O protocol that is independent of this block-level I/O protocol. For details, see Port-Level I/O Protocols.

6. Output `ap_done` goes High when the block completes operation.

   Note: If there is an `ap_return` port, the data on this port is valid when `ap_done` is High. Therefore, the `ap_done` signal also indicates when the data on output `ap_return` is valid.

7. When the design is ready to accept new inputs, the `ap_ready` signal goes High. Following is additional information about the `ap_ready` signal:
    - The `ap_ready` signal is inactive until the design starts operation.
    - In non-pipelined designs, the `ap_ready` signal is asserted at the same time as `ap_done` .
    - In pipelined designs, the `ap_ready` signal might go High at any cycle after `ap_start` is sampled High. This depends on how the design is pipelined.
    - If the `ap_start` signal is Low when `ap_ready` is High, the design executes until `ap_done` is High and then stops operation.
    - If the `ap_start` signal is High when `ap_ready` is High, the next transaction starts immediately, and the design continues to operate.
8. The `ap_idle` signal indicates when the design is idle and not operating. Following is additional information about the `ap_idle` signal:
    - If the `ap_start` signal is Low when `ap_ready` is High, the design stops operation, and the `ap_idle` signal goes High one cycle after `ap_done` .
    - If the `ap_start` signal is High when `ap_ready` is High, the design continues to operate, and the `ap_idle` signal remains Low.

ap_ctrl_chain

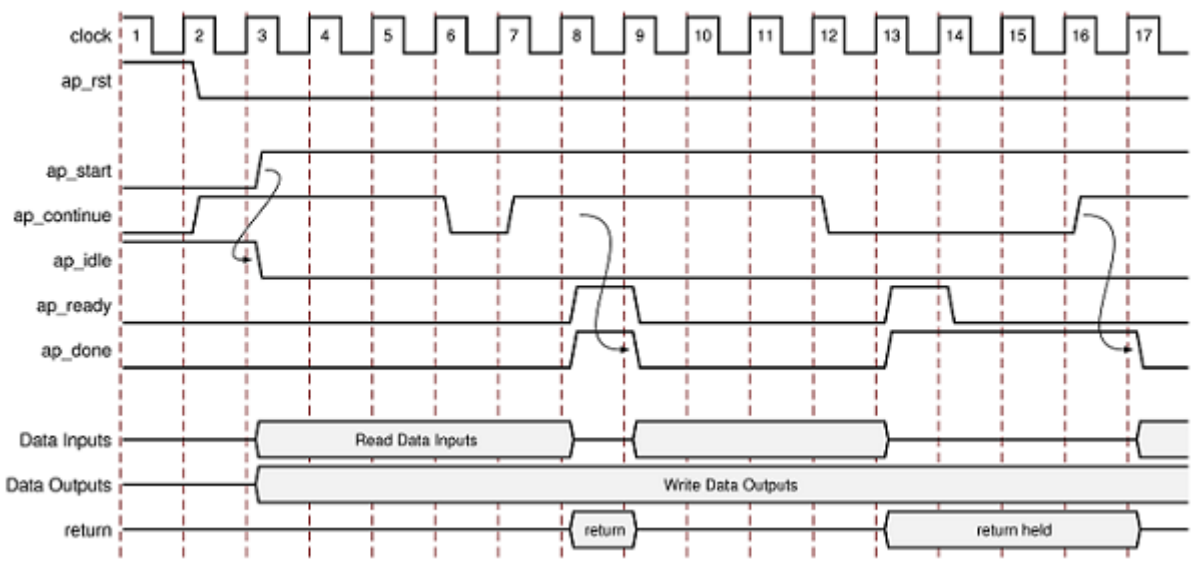
The `ap_ctrl_chain` block-level I/O protocol is similar to the `ap_ctrl_hs` protocol but provides an additional input port named `ap_continue` . An active High `ap_continue` signal indicates that the downstream block that consumes the output data is ready for new data inputs. If the downstream block is not able to consume new data inputs, the `ap_continue` signal is Low, which prevents upstream blocks from generating additional data.

The `ap_ready` port of the downstream block can directly drive the `ap_continue` port. Following is additional information about the `ap_continue` port:

- If the `ap_continue` signal is High when `ap_done` is High, the design continues operating. The behavior of the other block-level I/O signals is identical to those described in the `ap_ctrl_hs` block-level I/O protocol.
- If the `ap_continue` signal is Low when `ap_done` is High, the design stops operating, the `ap_done` signal remains High, and data remains valid on the `ap_return` port if the `ap_return` port is present.

In the following figure, the first transaction completes, and the second transaction starts immediately because `ap_continue` is High when `ap_done` is High. However, the design halts at the end of the second transaction until `ap_continue` is asserted High.

Figure 14: Behavior of ap_ctrl_chain Interface



Port-Level I/O Protocols

By default input pointers and pass-by-value arguments are implemented as simple wire ports with no associated handshaking signal. For example, in the `sum_io` function discussed in Interface Synthesis Overview, the input ports are implemented without an I/O protocol, only a data port. If the port has no I/O protocol, (by default or by design) the input data must be held stable until it is read.

By default output pointers are implemented with an associated output valid signal to indicate when the output data is valid. In the `sum_io` function example, the output port is implemented with an associated output valid port (`sum_o_ap_vld`) which indicates when the data on the port is valid and can be read. If there is no I/O protocol associated with the output port, it is difficult to know when to read the data.

> **TIP:** It is always a good idea to use an I/O protocol on an output.

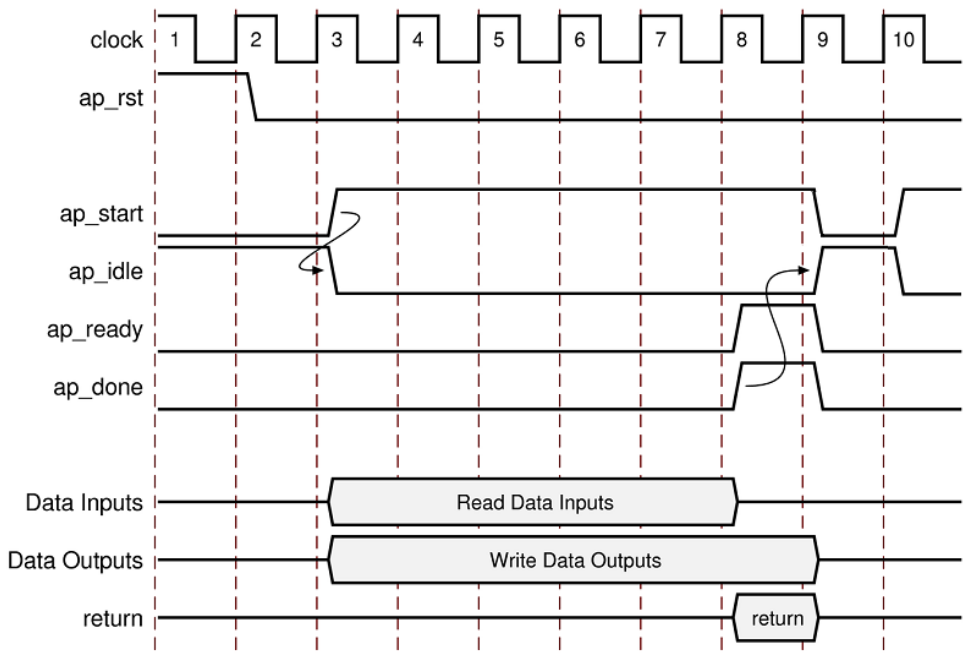
Function arguments which are both read from and written to are split into separate input and output ports. In the `sum_io` function example, the `sum` argument is implemented as both an input port `sum_i` , and an output port `sum_o` with associated I/O protocol port `sum_o_ap_vld` .

If the function has a return value, an output port `ap_return` is implemented to provide the return value. When the RTL design completes one transaction - this is equivalent to one execution of the C function - the block-level protocols indicate the function is complete with the `ap_done` signal. This also indicates the data on port `ap_return` is valid and can be read.

Note: The return value of the top-level function cannot be a pointer.

For the example code shown the timing behavior is shown in the following figure (assuming that the target technology and clock frequency allow a single addition per clock cycle).

Figure 15: RTL Port Timing with Default Synthesis



- The design starts when `ap_start` is asserted `High`.
- The `ap_idle` signal is asserted `Low` to indicate the design is operating.
- The input data is read at any clock after the first cycle. Vitis HLS schedules when the reads occur. The `ap_ready` signal is asserted high when all inputs have been read.
- When output `sum` is calculated, the associated output handshake (`sum_o_ap_vld`) indicates that the data is valid.
- When the function completes, `ap_done` is asserted. This also indicates that the data on `ap_return` is valid.
- Port `ap_idle` is asserted `High` to indicate that the design is waiting start again.

Port-Level I/O: AXI4 Interface Protocol

The AXI4 interfaces supported by Vitis HLS include the AXI4-Stream interface (`axis`), AXI4-Lite (`s_axilite`), and AXI4 master (`m_axi`) interfaces.

IMPORTANT: The AXI4 interfaces are the default port-level interfaces used by Vitis HLS when the Vitis kernel flow is selected (`open_solution -flow_target vitis`) as discussed in Creating a New Vitis HLS Project (creatingnewvitishlsproject.html#thg1583443745171).

For a complete description of the AXI4 interfaces, including timing and ports, see the *Vivado Design Suite: AXI Reference Guide* (UG1037 (https://www.xilinx.com/cgi-bin/docs/ipdoc?c=axi_ref_guide;v=latest;d=ug1037-vivado-axi-reference-guide.pdf)). For information on using the full capabilities of this I/O protocol, see Using AXI4 Interfaces.

axis

The `axis` mode specifies an AXI4-Stream I/O protocol. Specify this protocol on input arguments or output arguments only, not on input/output arguments.

s_axilite

The `s_axilite` mode specifies an AXI4-Lite slave I/O protocol. Specify this protocol on any type of argument except streams.

TIP: You can group multiple arguments into a single AXI4-Lite interface.

m_axi

The `m_axi` mode specifies an AXI4 master I/O protocol. Specify on arrays and pointers (and references in C++) only.

TIP: You can group multiple arguments into the same AXI4 interface.

Port-Level I/O: No Protocol

The `ap_none` and `ap_stable` modes specify that no I/O protocol be added to the port. When these modes are specified the argument is implemented as a data port with no other associated signals. The `ap_none` mode is the default for scalar inputs. The `ap_stable` mode is intended for configuration inputs that only change when the device is in reset mode.

ap_none

The `ap_none` port-level I/O protocol is the simplest interface type and has no other signals associated with it. Neither the input nor output data signals have associated control ports that indicate when data is read or written. The only ports in the RTL design are those specified in the source code.

An `ap_none` interface does not require additional hardware overhead. However, the `ap_none` interface does requires the following:

- Producer blocks to do one of the following:
  - Provide data to the input port at the correct time
  - Hold data for the length of a transaction until the design completes
- Consumer blocks to read output ports at the correct time

Note: The `ap_none` interface cannot be used with array arguments.

ap_stable

Like `ap_none`, the `ap_stable` port-level I/O protocol does not add any interface control ports to the design. The `ap_stable` type is typically used for data that can change but remains stable during normal operation, such as ports that provide configuration data. The `ap_stable` type informs Vitis HLS of the following:

- The data applied to the port remains stable during normal operation but is not a constant value that can be optimized.
- The fanout from this port is not required to be registered.

Note: The `ap_stable` type can only be applied to input ports. When applied to inout ports, only the input of the port is considered stable.

Port-Level I/O: Wire Handshakes

Interface mode `ap_hs` includes a two-way handshake signal with the data port. The handshake is an industry standard valid and acknowledge handshake. Mode `ap_vld` is the same but only has a valid port and `ap_ack` only has a acknowledge port.

Mode `ap_ovld` is for use with in-out arguments. When the in-out is split into separate input and output ports, mode `ap_none` is applied to the input port and `ap_vld` applied to the output port. This is the default for pointer arguments that are both read and written.

The `ap_hs` mode can be applied to arrays that are read or written in sequential order. If Vitis HLS can determine the read or write accesses are not sequential, it will halt synthesis with an error. If the access order cannot be determined, Vitis HLS will issue a warning.

ap_hs (ap_ack, ap_vld, and ap_ovld)

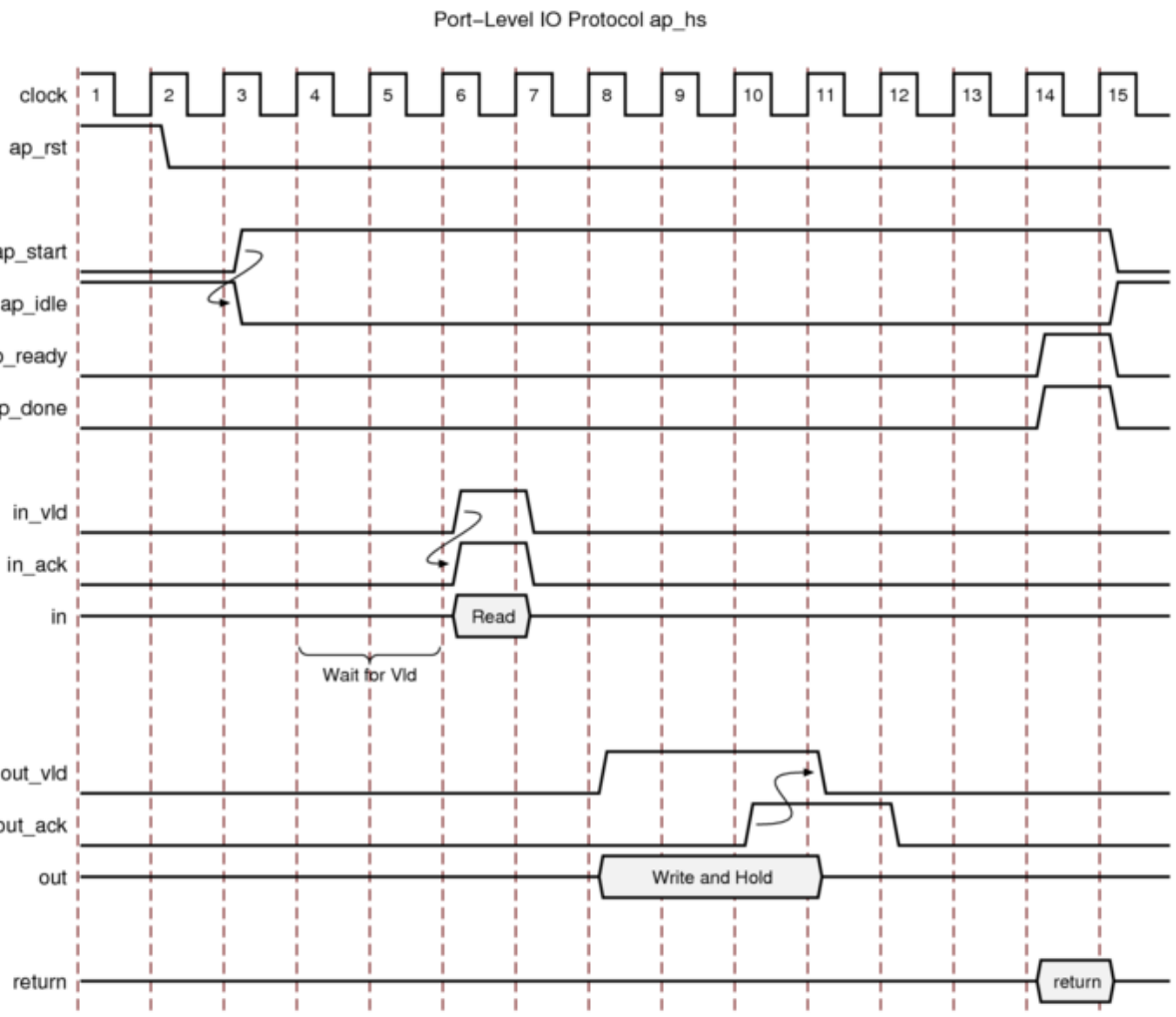
The `ap_hs` port-level I/O protocol provides the greatest flexibility in the development process, allowing both bottom-up and top-down design flows. Two-way handshakes safely perform all intra-block communication, and manual intervention or assumptions are not required for correct operation. The `ap_hs` port-level I/O protocol provides the following signals:

- Data port
- Acknowledge signal to indicate when data is consumed.
- Valid signal to indicate when data is read.

The following figure shows how an `ap_hs` interface behaves for both an input and output port. In this example, the input port is named `in`, and the output port is named `out`.

Note: The control signals names are based on the original port name. For example, the valid port for data input `in` is named `in_vld`.

Figure 16: Behavior of ap_hs Interface



For inputs, the following occurs:

- After start is applied, the block begins normal operation.
- If the design is ready for input data but the input valid is Low, the design stalls and waits for the input valid to be asserted to indicate a new input value is present.

  Note: The preceding figure shows this behavior. In this example, the design is ready to read data input in on clock cycle 4 and stalls waiting for the input valid before reading the data.

- When the input valid is asserted High, an output acknowledge is asserted High to indicate the data was read.

For outputs, the following occurs:

- After start is applied, the block begins normal operation.
- When an output port is written to, its associated output valid signal is simultaneously asserted to indicate valid data is present on the port.
- If the associated input acknowledge is Low, the design stalls and waits for the input acknowledge to be asserted.
- When the input acknowledge is asserted, the output valid is deasserted on the next clock edge.

ap_ack

The `ap_ack` port-level I/O protocol is a subset of the `ap_hs` interface type. The `ap_ack` port-level I/O protocol provides the following signals:

- Data port
- Acknowledge signal to indicate when data is consumed
  - For input arguments, the design generates an output acknowledge that is active-High in the cycle the input is read.
  - For output arguments, Vitis HLS implements an input acknowledge port to confirm the output was read.

  Note: After a write operation, the design stalls and waits until the input acknowledge is asserted High, which indicates the output was read by a consumer block. However, there is no associated output port to indicate when the data can be consumed.

CAUTION: You cannot use C/RTL co-simulation to verify designs that use `ap_ack` on an output port.

ap_vld

The `ap_vld` is a subset of the `ap_hs` interface type. The `ap_vld` port-level I/O protocol provides the following signals:

- Data port
- valid signal to indicate when data is read
  - For input arguments, the design reads the data port as soon as the valid is active. Even if the design is not ready to read new data, the design samples the data port and holds the data internally until needed.
  - For output arguments, Vitis HLS implements an output valid port to indicate when the data on the output port is valid.

ap_ovld

The `ap_ovld` is a subset of the `ap_hs` interface type. The `ap_ovld` port-level I/O protocol provides the following signals:

- Data port
- valid signal to indicate when data is read
    - For input arguments and the input half of inout arguments, the design defaults to type `ap_none` .
    - For output arguments and the output half of inout arguments, the design implements type `ap_vld` .

Port-Level I/O: Memory Interface Protocol

Array arguments are implemented by default as an `ap_memory` interface. This is a standard block RAM interface with data, address, chip-enable, and write-enable ports.

An `ap_memory` interface may be implemented as a single-port of dual-port interface. If Vitis HLS can determine that using a dual-port interface will reduce the initial interval, it will automatically implement a dual-port interface. The BIND_STORAGE pragma or directive is used to specify the memory resource and if this directive is specified on the array with a single-port block RAM, a single-port interface will be implemented. Conversely, if a dual-port interface is specified using the BIND_STORAGE pragma and Vitis HLS determines this interface provides no benefit it will automatically implement a single-port interface.

The `bram` interface mode is functional identical to the `ap_memory` interface. The only difference is how the ports are implemented when the design is used in Vitis IP Integrator:

- An `ap_memory` interface is displayed as multiple and separate ports.
- A `bram` interface is displayed as a single grouped port which can be connected to a Xilinx block RAM using a single point-to-point connection.

If the array is accessed in a sequential manner an `ap_fifo` interface can be used. As with the `ap_hs` interface, Vitis HLS will halt if it determines the data access is not sequential, report a warning if it cannot determine if the access is sequential or issue no message if it determines the access is sequential. The `ap_fifo` interface can only be used for reading or writing, not both.

ap_memory, bram

The `ap_memory` and `bram` interface port-level I/O protocols are used to implement array arguments. This type of port-level I/O protocol can communicate with memory elements (for example, RAMs and ROMs) when the implementation requires random accesses to the memory address locations.

Note: If you only need sequential access to the memory element, use the `ap_fifo` interface instead. The `ap_fifo` interface reduces the hardware overhead, because address generation is not performed.

The `ap_memory` and `bram` interface port-level I/O protocols are identical. The only difference is the way Vivado IP integrator shows the blocks:

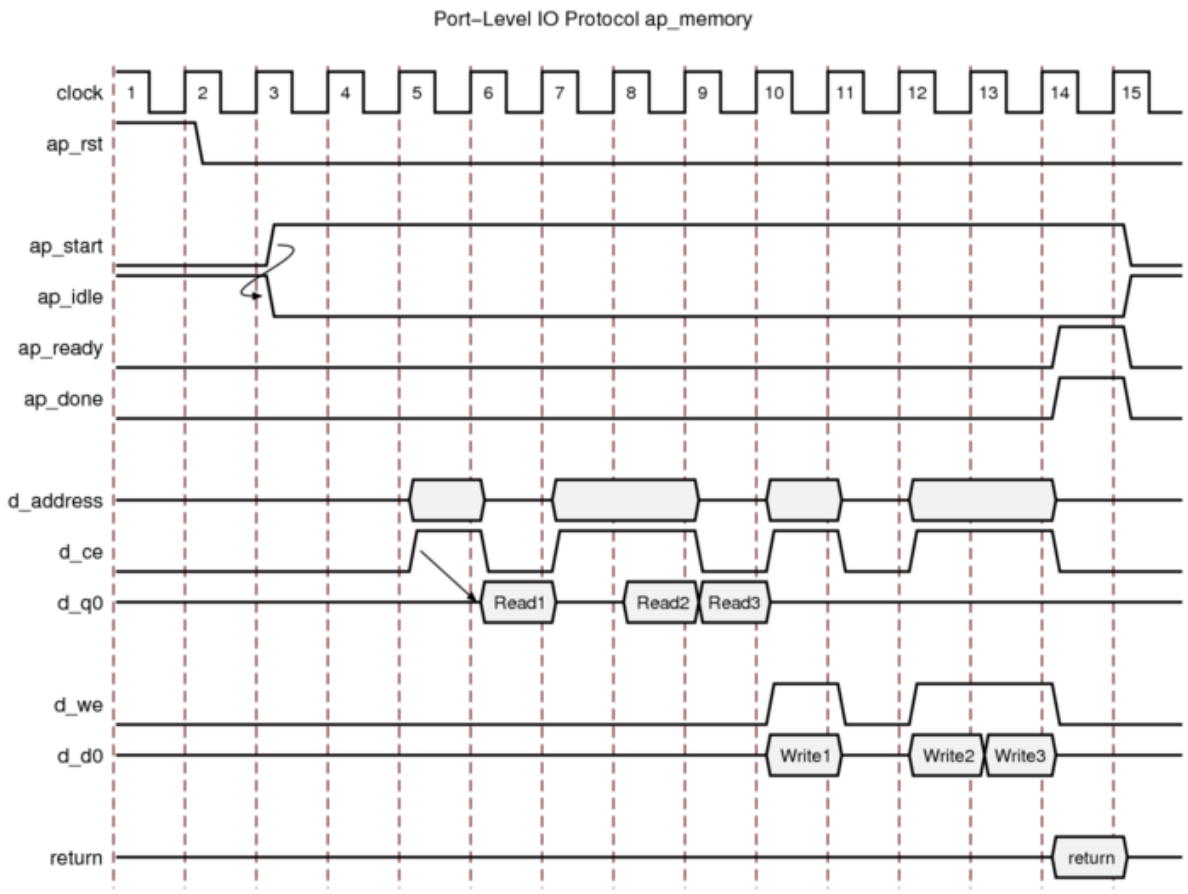
- The `ap_memory` interface appears as discrete ports.
- The `bram` interface appears as a single, grouped port. In IP integrator, you can use a single connection to create connections to all ports.

When using an `ap_memory` interface, specify the array targets using the BIND_STORAGE pragma. If no target is specified for the arrays, Vitis HLS determines whether to use a single or dual-port RAM interface.

TIP: Before running synthesis, ensure array arguments are targeted to the correct memory type using the BIND_STORAGE pragma. Re-synthesizing with corrected memories can result in a different schedule and RTL.

The following figure shows an array named `d` specified as a single-port block RAM. The port names are based on the C function argument. For example, if the C argument is `d` , the chip-enable is `d_ce` , and the input data is `d_q0` based on the `output/q` port of the BRAM.

Figure 17: Behavior of ap_memory Interface



After reset, the following occurs:

- After start is applied, the block begins normal operation.
- Reads are performed by applying an address on the output address ports while asserting the output signal `d_ce`.

  Note: For a default block RAM, the design expects the input data `d_q0` to be available in the next clock cycle. You can use the BIND_STORAGE pragma to indicate the RAM has a longer read latency.

- Write operations are performed by asserting output ports `d_ce` and `d_we` while simultaneously applying the address and output data `d_d0`.

ap_fifo

When an output port is written to, its associated output valid signal interface is the most hardware-efficient approach when the design requires access to a memory element and the access is always performed in a sequential manner, that is, no random access is required. The `ap_fifo` port-level I/O protocol supports the following:

- Allows the port to be connected to a FIFO
- Enables complete, two-way `empty-full` communication
- Works for arrays, pointers, and pass-by-reference argument types

Note: Functions that can use an `ap_fifo` interface often use pointers and might access the same variable multiple times. To understand the importance of the `volatile` qualifier when using this coding style, see Multi-Access Pointer Interfaces: Streaming Data.

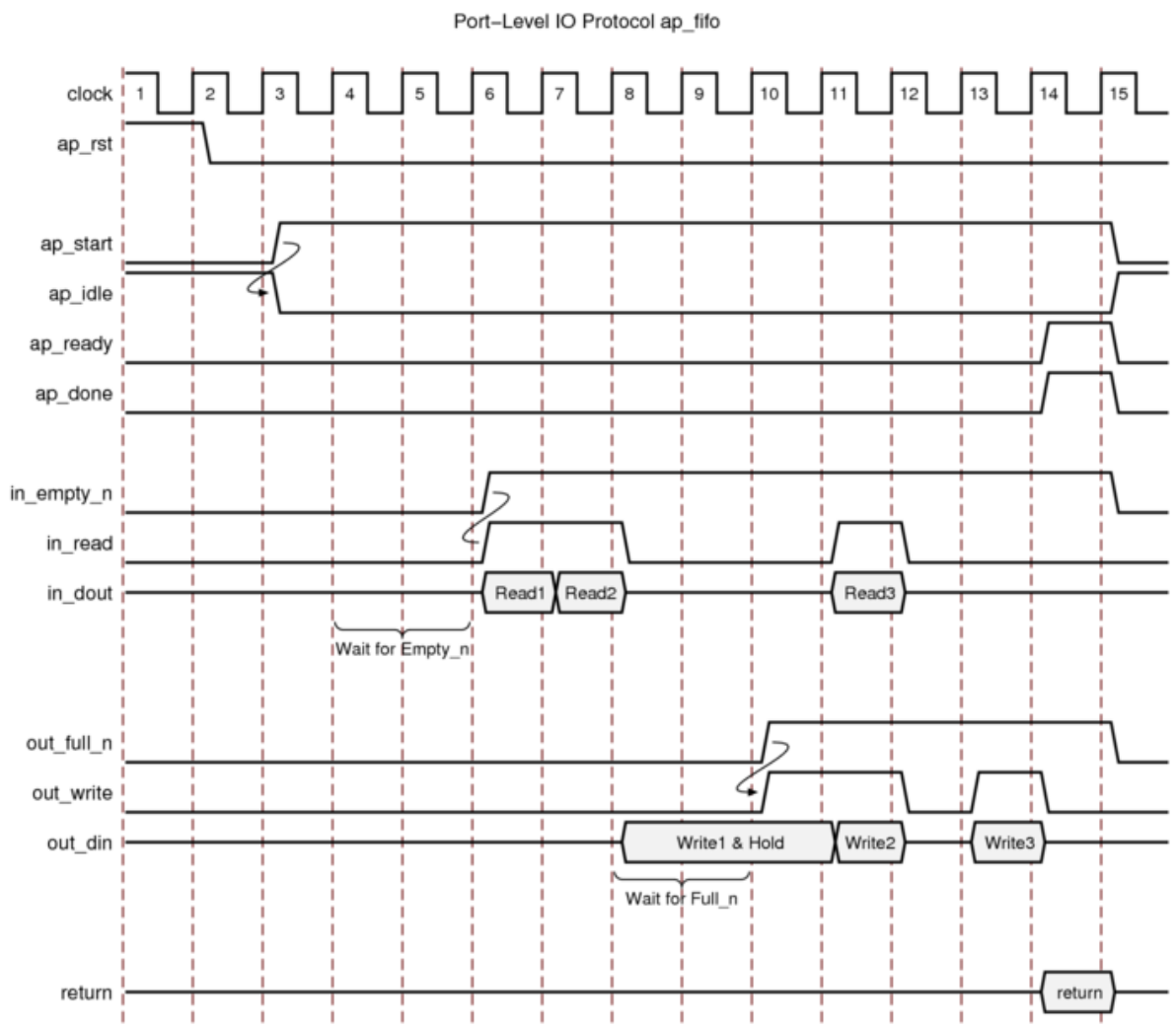
In the following example, `in1` is a pointer that accesses the current address, then two addresses above the current address, and finally one address below.

```
void foo(int* in1, ...) {
 int data1, data2, data3;
       ...
 data1= *in1;
 data2= *(in1+2);
 data3= *(in1-1);
 ...
}
```

If `in1` is specified as an `ap_fifo` interface, Vitis HLS checks the accesses, determines the accesses are not in sequential order, issues an error, and halts. To read from non-sequential address locations, use an `ap_memory` or `bram` interface.

You cannot specify an `ap_fifo` interface on an argument that is both read from and written to. You can only specify an `ap_fifo` interface on an input or an output argument. A design with input argument `in` and output argument `out` specified as `ap_fifo` interfaces behaves as shown in the following figure.

Figure 18: Behavior of ap_fifo Interface



For inputs, the following occurs:

- After start is applied, the block begins normal operation.
- If the input port is ready to be read but the FIFO is empty as indicated by input port `in_empty_n` Low, the design stalls and waits for data to become available.
- When the FIFO contains data as indicated by input port `in_empty_n` High, an output acknowledge `in_read` is asserted High to indicate the data was read in this cycle.

For outputs, the following occurs:

- After start is applied, the block begins normal operation.
- If an output port is ready to be written to but the FIFO is full as indicated by `out_full_n` Low, the data is placed on the output port but the design stalls and waits for the space to become available in the FIFO.
- When space becomes available in the FIFO as indicated by `out_full_n` High, the output acknowledge signal `out_write` is asserted to indicate the output data is valid.
- If the top-level function or the top-level loop is pipelined using the `-rewind` option, Vitis HLS creates an additional output port with the suffix `_lwr` . When the last write to the FIFO interface completes, the `_lwr` port goes active-High.

Interface Synthesis and Structs

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. As part of aggregation, the elements of the struct are also aligned on a 4 byte alignment, and this alignment might require the use of bit padding to keep things aligned, as discussed in Struct Padding and Alignment.

TIP: Structs are aggregated by default in the interface, but they are not packed. If you want the struct packed, you must use `__attribute__(packed)` .

The member elements of the struct are placed into the vector in the order they appear in the 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.

IMPORTANT: Structs on the interface are aggregated by default, and can not be disaggregated. To disaggregate a struct on the interface, you must manually code it as separate elements.

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 24-bit port containing 16 bits of `varA` , and 8 bits of `varB` .

TIP: The maximum bit-width of any port or bus created by data packing is 8192 bits.

If a struct contains an array, aggregating the struct performs an operation similar to ARRAY_RESHAPE, and then combines the reshaped array with other elements of the struct. A packed struct cannot be optimized with ARRAY_PARTITION, or ARRAY_RESHAPE.

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).

Interface Synthesis and Multi-Access Pointers

Using pointers which are accessed multiple times can introduce unexpected behavior after synthesis. In the following 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 compiler, Vitis HLS will optimize away the redundant pointer accesses. To implement the above code 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 next 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;
}
```

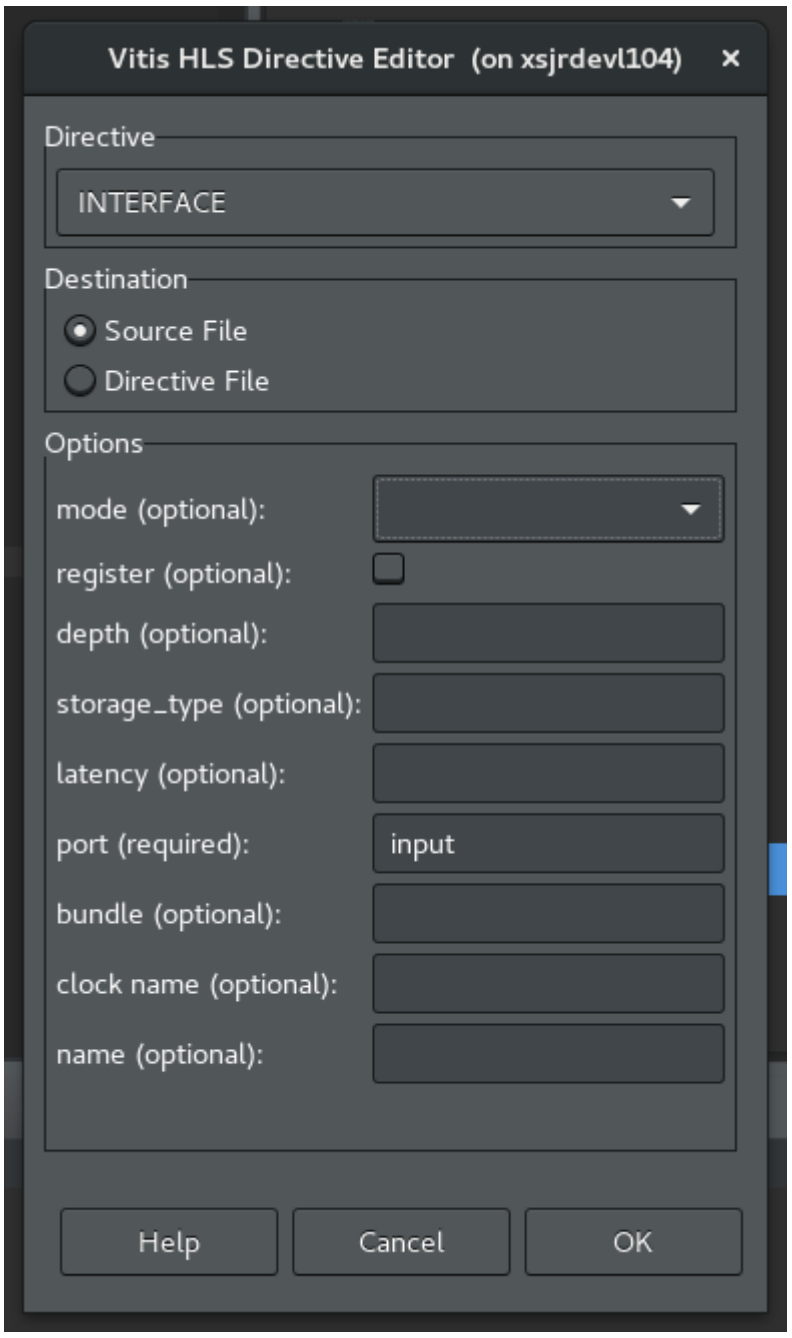
Even this C code is problematic. Indeed, using a test bench, there is no way to supply anything but a single value to `d_i` or verify any write to `d_o` other than the final write. Although multi-access pointers are supported, it is highly recommended to implement the behavior required using the `hls::stream` class. Details on the `hls::stream` class are in HLS Stream Library (vitishlslibrariesreference.html#mes1539734221433).

Specifying Interfaces

Interface synthesis is controlled by default by the configuration settings defined by the config_interface (lls1585342611108.html) command, as well as by the type of arguments in your top-level function. You can change the defaults defined in the configuration settings by opening the **Solution Settings** dialog box as described in Setting Configuration Options (creatingnewvitishlsproject.html#zle1584656100548).

The INTERFACE pragma or directive lets you specify details for a specific function argument, or interface port, augmenting the default configuration or overriding it as needed. To specify the interface mode for ports, right-click the port or argument on the top-level function in the **Directives** tab in the Vitis HLS GUI, and select **Insert Directive** to open the **Vitis HLS Directive Editor** as shown in the following figure.

Figure 19: Set Directive: Interface



The options for the INTERFACE directive change depending on the specific interface mode you select. Refer to set_directive_interface (knt1585343372285.html) for details on the various options, or select the **Help** command in the Directive Editor dialog box. Following are some options of interest:

- **mode**

  Select the interface mode from the drop-down menu. The choices presented by the Directive Editor change depending on the specific interface mode you choose for the port.

- **register**

  If you select this option, all pass-by-value reads are performed in the first cycle of operation. For output ports, the register option guarantees the output is registered. You can apply the register option to any function in the design. For memory, FIFO, and AXI4 interfaces, the register option has no effect.

- **port**

  This option is required. By default, Vitis HLS does not register ports.

- **depth**

  This option specifies how many samples are provided to the design by the test bench and how many output values the test bench must store. Use whichever number is greater.

  Note: For cases in which a pointer is read from or written to multiple times within a single transaction, the **depth** option is required for C/RTL co-simulation. The **depth** option is *not* required for arrays or when using the `hls::stream` construct. It is only required when using pointers on the interface.

  If the **depth** option is set too small, the C/RTL co-simulation might deadlock as follows:

  1. The input reads might stall waiting for data that the test bench cannot provide.
  2. The output writes might stall when trying to write data, because the storage is full.

- **offset**

  This option is used for AXI4 interfaces.

Using AXI4 Interfaces

AXI4-Stream Interfaces

An AXI4-Stream interface can be applied to any input argument and any array or pointer output argument. Since an AXI4-Stream interface transfers data in a sequential streaming manner it cannot be used with arguments that are both read and written. An AXI4-Stream interface is always sign-extended to the next byte. For example, a 12-bit data value is sign-extended to 16-bit.

AXI4-Stream interfaces are always implemented as registered interfaces to ensure no combinational feedback paths are created when multiple HLS IP blocks with AXI-Stream interfaces are integrated into a larger design. For AXI-Stream interfaces, four types of register modes are provided to control how the AXI-Stream interface registers are implemented.

- Forward: Only the `TDATA` and `TVALID` signals are registered.
- Reverse: Only the `TREADY` signal is registered.
- Both: All signals (`TDATA` , `TREADY` and `TVALID`) are registered. This is the default.
- Off: None of the port signals are registered.

The AXI-Stream side-channel signals are considered to be data signals and are registered whenever TDATA is registered.

Note: When connecting HLS generated IP blocks with AXI4-Stream interfaces at least one interface should be implemented as a registered interface or the blocks should be connected via an AXI4-Stream Register Slice.

There are two basic ways to use an AXI4-Stream in your design.

- Use an AXI4-Stream without side-channels.
- Use an AXI4-Stream with side-channels.

This second use model provides additional functionality, allowing the optional side-channels which are part of the AXI4-Stream standard, to be used directly in the C code.

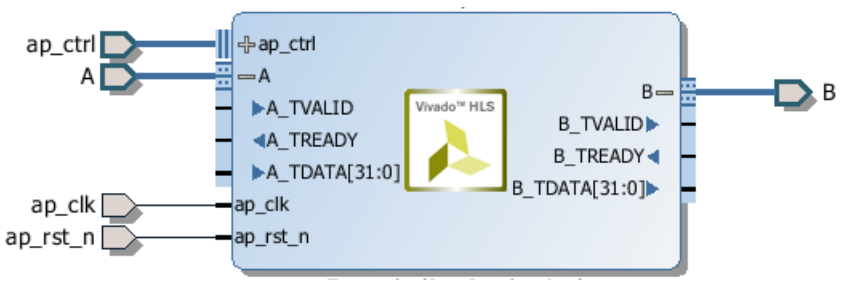
AXI4-Stream Interfaces without Side Channels

An AXI4-Stream is used without side-channels when the function argument, `ap_axis` data type, does not contain any AXI4 side-channel elements. In this example, both interfaces are implemented using an AXI4-Stream.

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


void example(hls::stream< ap_axis<32>> &A,
         hls::stream< ap_axis<32>> &B)
{
#pragma HLS INTERFACE axis port=A
#pragma HLS INTERFACE axis port=B

    ap_axis<32> tmp;

    A.read(tmp);
    tmp.data = tmp + 5;
    B.write(tmp);
}
```

After synthesis, both arguments are implemented with a data port and the standard AXI4-Stream TVALID and TREADY protocol ports as shown in the following figure.

Figure 20: AXI4-Stream Interfaces Without Side-Channels



Multiple variables can be combined into the same AXI4-Stream interface by using a struct, which is aggregated by Vitis HLS by default. Aggregating the elements of a struct into a single wide-vector, allows all elements of the struct to be implemented in the same AXI4-Stream interface.

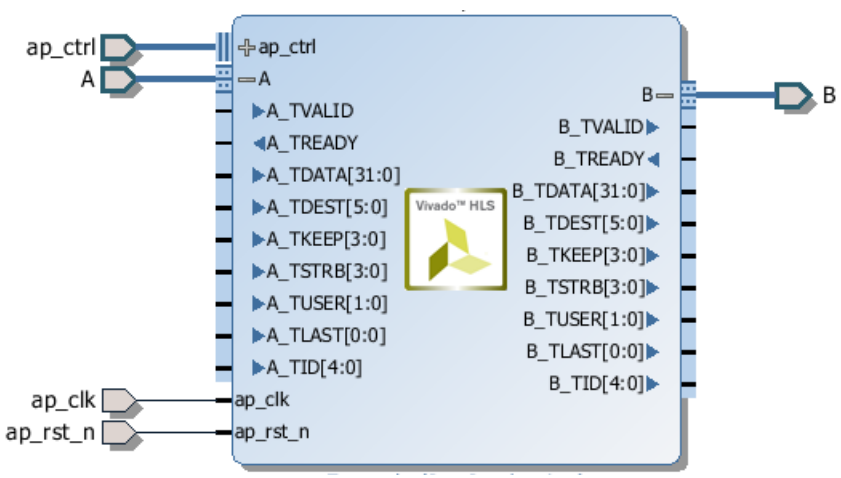
AXI4-Stream Interfaces with Side-Channels

The following example shows how the side-channels can be used directly in the C code and implemented on the interface. The code uses the `ap_axi_sdata.h` include file for an API to handle the side channels of the AXI4-Stream interface. In this example a signed 32-bit data type is used.

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

#define DWIDTH 32

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

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

   bool eos = false;
   vmult:
      do {
         #pragma HLS PIPLINE II=1
         trans_pkt t2 = b.read();

         //Patcket for Output
         trans_pkt t_out;

         // Reading data from input packet
         ap_uint<DWIDTH> in2 = t2.data;
         ap_uint<DWIDTH> tmpOut = in2 * 5;

         // Setting data and configuration to output packet
         t_out.data = tmpOut;
         t_out.last = t1.get_last();
         t_out.keep = -1; //Enabling all bytes

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

After synthesis, both arguments are implemented with data ports, the standard AXI4-Stream `TVALID` and `TREADY` protocol ports and all of the optional ports described in the struct.

Figure 21: AXI4-Stream Interfaces With Side-Channels



Packing Structs into AXI4-Stream Interfaces

There is a difference in the default synthesis behavior when using structs with AXI4-Stream interfaces. The default synthesis behavior for struct is described in Interface Synthesis and Structs.

When using AXI4-Stream interfaces with or without side-channels and the function argument is a struct, Vitis HLS automatically packs all elements of the struct into a single wide-data vector. The interface is implemented as a single wide-data vector with associated `TVALID` and `TREADY` signals.

When using AXI4-Stream interfaces with side-channels, the Vitis HLS tool does not let you send a struct, such as RGBPixel, across the axis interface. You cannot overload the `ap_axiu` struct, to define your own struct. The workaround to this limitation is to use the struct initialize function, to get the intended result.

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

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

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

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

AXI4-Lite Interface

You can use an AXI4-Lite interface to allow the design to be controlled by a CPU or microcontroller. Using the Vitis HLS AXI4-Lite interface, you can:

- Group multiple ports into the same AXI4-Lite interface.
- Output C driver files for use with the code running on a processor.

Note: This provides a set of C application program interface (API) functions, which allows you to easily control the hardware from the software. This is useful when the design is exported to the IP Catalog.

The following example shows how Vitis HLS implements multiple arguments, including the function return, as an AXI4-Lite interface. Because each directive uses the same name for the `bundle` option, each of the ports is grouped into the same AXI4-Lite interface.

```
void example(char *a, char *b, char *c)
{
#pragma HLS INTERFACE s_axilite port=return bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=a      bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=b      bundle=BUS_A
#pragma HLS INTERFACE s_axilite port=c      bundle=BUS_A offset=0x0400
#pragma HLS INTERFACE ap_vld port=b

  *c += *a + *b;
}
```

Note: If you do not use the `bundle` option, Vitis HLS groups all arguments specified with an AXI4-Lite interface into the same default bundle and automatically names the port.

You can also assign an I/O protocol to ports grouped into an AXI4-Lite interface. In the example above, Vitis HLS implements port b as an `ap_vld` interface and groups port `b` into the AXI4-Lite interface. As a result, the AXI4-Lite interface contains a register for the port `b` data, a register for the output to acknowledge that port `b` was read, and a register for the port `b` input valid signal.

Each time port `b` is read, Vitis HLS automatically clears the input valid register and resets the register to logic 0. If the input valid register is not set to logic 1, the data in the `b` data register is not considered valid, and the design stalls and waits for the valid register to be set.

Note: For ease of use during the operation of the design, Xilinx recommends that you do not include additional I/O protocols in the ports grouped into an AXI4-Lite interface. However, Xilinx recommends that you include the block-level I/O protocol associated with the `return` port in the AXI4-Lite interface.

You cannot assign arrays to an AXI4-Lite interface using the `bram` interface. You can only assign arrays to an AXI4-Lite interface using the default `ap_memory` interface. You also cannot assign any argument specified with `ap_stable` I/O protocol to an AXI4-Lite interface.

Since the variables grouped into an AXI4-Lite interface are function arguments, which themselves cannot be assigned a default value in the C code, none of the registers in an AXI4-Lite interface may be assigned a default value. The registers can be implemented with a reset with the `config_rtl` command, but they cannot be assigned any other default value.

By default, Vitis HLS automatically assigns the address for each port that is grouped into an AXI4-Lite interface. Vitis HLS provides the assigned addresses in the C driver files. For more information, see C Driver Files. To explicitly define the address, you can use the `offset` option, as shown for argument `c` in the example above.

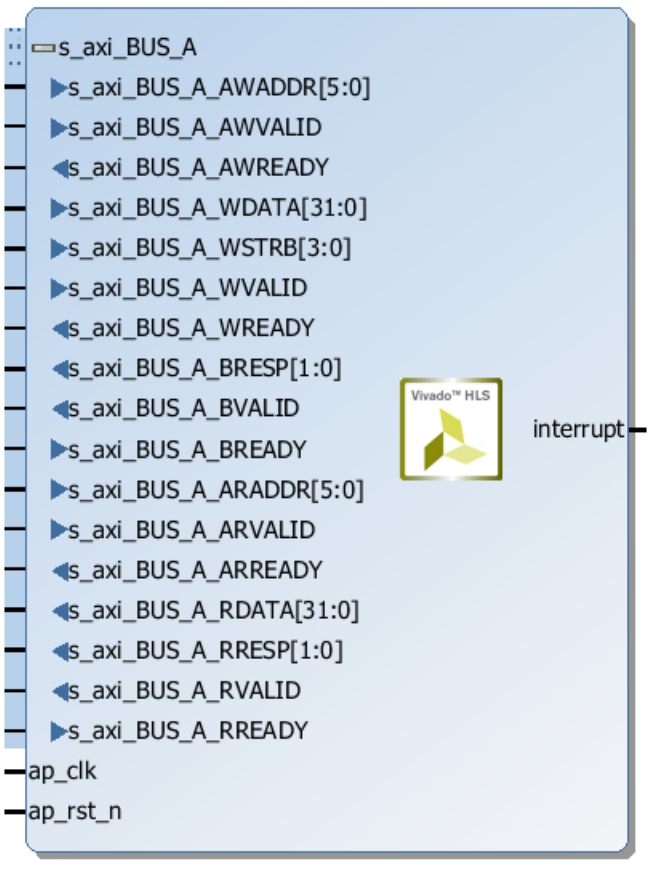
IMPORTANT: In an AXI4-Lite interface, Vitis HLS reserves addresses 0x0000 through 0x000C for the block-level I/O protocol signals and interrupt controls.

After synthesis, Vitis HLS implements the ports in the AXI4-Lite port, as shown in the following figure. Vitis HLS creates the interrupt port by including the function return in the AXI4-Lite interface. You can program the interrupt through the AXI4-Lite interface. You can also drive the interrupt from the following block-level protocols:

- `ap_done` : Indicates when the function completes all operations.
- `ap_ready` : Indicates when the function is ready for new input data.

You can program the interface using the C driver files.

Figure 22: AXI4-Lite Slave Interfaces with Grouped RTL Ports



Control Clock and Reset in AXI4-Lite Interfaces

By default, Vitis HLS uses the same clock for the AXI4-Lite interface and the synthesized design. Vitis HLS connects all registers in the AXI4-Lite interface to the clock used for the synthesized logic (`ap_clk`).

Optionally, you can use the INTERFACE directive `clock` option to specify a separate clock for each AXI4-Lite port. When connecting the clock to the AXI4-Lite interface, you must use the following protocols:

- AXI4-Lite interface clock must be synchronous to the clock used for the synthesized logic (`ap_clk`). That is, both clocks must be derived from the same master generator clock.
- AXI4-Lite interface clock frequency must be equal to or less than the frequency of the clock used for the synthesized logic (`ap_clk`).

If you use the `clock` option with the interface directive, you only need to specify the `clock` option on one function argument in each bundle. Vitis HLS implements all other function arguments in the bundle with the same clock and reset. Vitis HLS names the generated reset signal with the prefix `ap_rst_` followed by the clock name. The generated reset signal is active Low independent of the `config_rtl` command.

The following example shows how Vitis HLS groups function arguments `a` and `b` into an AXI4-Lite port with a clock named `AXI_clk1` and an associated reset port.

```
// Default AXI-Lite interface implemented with independent clock called AXI_clk1
#pragma HLS interface s_axilite port=a clock=AXI_clk1
#pragma HLS interface s_axilite port=b
```

In the following example, Vitis HLS groups function arguments `c` and `d` into AXI4-Lite port `CTRL1` with a separate clock called `AXI_clk2` and an associated reset port.

```
// CTRL1 AXI-Lite bundle implemented with a separate clock (called AXI_clk2)
#pragma HLS interface s_axilite port=c bundle=CTRL1 clock=AXI_clk2
#pragma HLS interface s_axilite port=d bundle=CTRL1
```

C Driver Files

When an AXI4-Lite slave interface is implemented, a set of C driver files are automatically created. These C driver files provide a set of APIs that can be integrated into any software running on a CPU and used to communicate with the device via the AXI4-Lite slave interface.

The C driver files are created when the design is packaged as IP in the IP Catalog.

Driver files are created for standalone and Linux modes. In standalone mode the drivers are used in the same way as any other Xilinx standalone drivers. In Linux mode, copy all the C files (.c) and header files (.h) files into the software project.

The driver files and API functions derive their name from the top-level function for synthesis. In the above example, the top-level function is called “example”. If the top-level function was named “DUT” the name “example” would be replaced by “DUT” in the following description. The driver files are created in the packaged IP (located in the `impl` directory inside the `solution`).

Table 6. C Driver Files for a Design Named `example`

File Path	Usage Mode	Description
data/example.mdd	Standalone	Driver definition file.
data/example.tcl	Standalone	Used by SDK to integrate the software into an SDK project.
src/xexample_hw.h	Both	Defines address offsets for all internal registers.
src/xexample.h	Both	API definitions
src/xexample.c	Both	Standard API implementations
src/xexample_sinit.c	Standalone	Initialization API implementations
src/xexample_linux.c	Linux	Initialization API implementations
src/Makefile	Standalone	Makefile

In file `xexample.h`, two structs are defined.

XExample_Config
: This is used to hold the configuration information (base address of each AXI4-Lite slave interface) of the IP instance.

XExample
: This is used to hold the IP instance pointer. Most APIs take this instance pointer as the first argument.

The standard API implementations are provided in files `xexample.c`, `xexample_sinit.c`, `xexample_linux.c`, and provide functions to perform the following operations.

- Initialize the device
- Control the device and query its status
- Read/write to the registers
- Set up, monitor, and control the interrupts

Refer to AXI4-Lite Slave C Driver Reference (axi4liteslavecdriverreference.html#rgw1539734256874) for a description of the API functions provided in the C driver files.

IMPORTANT: The C driver APIs always use an unsigned 32-bit type (U32). You might be required to cast the data in the C code into the expected type.

C Driver Files and Float Types

C driver files always use a data 32-bit unsigned integer (U32) for data transfers. In the following example, the function uses float type arguments `a` and `r1`. It sets the value of `a` and returns the value of `r1`:

```
float caculate(float a, float *r1)
{
#pragma HLS INTERFACE ap_vld register port=r1
#pragma HLS INTERFACE s_axilite port=a
#pragma HLS INTERFACE s_axilite port=r1
#pragma HLS INTERFACE s_axilite port=return

 *r1 = 0.5f*a;
 return (a>0);
}
```

After synthesis, Vitis HLS groups all ports into the default AXI4-Lite interface and creates C driver files. However, as shown in the following example, the driver files use type U32:

```
// API to set the value of A
void XCaculate_SetA(XCaculate *InstancePtr, u32 Data) {
    Xil_AssertVoid(InstancePtr != NULL);
    Xil_AssertVoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);
    XCaculate_WriteReg(InstancePtr->Hls_periph_bus_BaseAddress,
XCACULATE_HLS_PERIPH_BUS_ADDR_A_DATA, Data);
}

// API to get the value of R1
u32 XCaculate_GetR1(XCaculate *InstancePtr) {
    u32 Data;

    Xil_AssertNonvoid(InstancePtr != NULL);
    Xil_AssertNonvoid(InstancePtr->IsReady == XIL_COMPONENT_IS_READY);

    Data = XCaculate_ReadReg(InstancePtr->Hls_periph_bus_BaseAddress,
XCACULATE_HLS_PERIPH_BUS_ADDR_R1_DATA);
    return Data;
}
```

If these functions work directly with float types, the write and read values are not consistent with expected float type. When using these functions in software, you can use the following casts in the code:

```
float a=3.0f,r1;
u32 ua,ur1;

// cast float “a” to type U32
XCaculate_SetA(&calculate,*((u32*)&a));
ur1=XCaculate_GetR1(&caculate);

// cast return type U32 to float type for “r1”
r1=*((float*)&ur1);
```

Controlling Hardware

The hardware header file `xexample_hw.h` (in this example) provides a complete list of the memory mapped locations for the ports grouped into the AXI4-Lite slave interface.

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/SC)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        others - reserved
// 0x10 : Data signal of a
//        bit 7~0 - a[7:0] (Read/Write)
//        others  - reserved
// 0x14 : reserved
// 0x18 : Data signal of b
//        bit 7~0 - b[7:0] (Read/Write)
//        others  - reserved
// 0x1c : reserved
// 0x20 : Data signal of c_i
//        bit 7~0 - c_i[7:0] (Read/Write)
//        others  - reserved
// 0x24 : reserved
// 0x28 : Data signal of c_o
//        bit 7~0 - c_o[7:0] (Read)
//        others  - reserved
// 0x2c : Control signal of c_o
//        bit 0  - c_o_ap_vld (Read/COR)
//        others - reserved
```

To correctly program the registers in the AXI4-Lite slave interface, there is some requirement to understand how the hardware ports operate. The block will operate with the same port protocols described in Interface Synthesis I/O Protocols.

For example, to start the block operation the `ap_start` register must be set to 1. The device will then proceed and read any inputs grouped into the AXI4-Lite slave interface from the register in the interface. When the block completes operation, the `ap_done`, `ap_idle` and `ap_ready` registers will be set by the hardware output ports and the results for any

output ports grouped into the AXI4-Lite slave interface read from the appropriate register.

The implementation of function argument `c` in the example above also highlights the importance of some understanding how the hardware ports are operate. Function argument `c` is both read and written to, and is therefore implemented as separate input and output ports `c_i` and `c_o`, as explained in Interface Synthesis Overview.

The first recommended flow for programing the AXI4-Lite slave interface is for a one-time execution of the function:

- Use the interrupt function to determine how you wish the interrupt to operate.
- Load the register values for the block input ports. In the above example this is performed using API functions `XExample_Set_a`, `XExample_Set_b`, and `XExample_Set_c_i`.
- Set the `ap_start` bit to 1 using `XExample_Start` to start executing the function. This register is self-clearing as noted in the header file above. After one transaction, the block will suspend operation.
- Allow the function to execute. Address any interrupts which are generated.
- Read the output registers. In the above example this is performed using API functions `XExample_Get_c_o_vld`, to confirm the data is valid, and `XExample_Get_c_o`.

  Note: The registers in the AXI4-Lite slave interface obey the same I/O protocol as the ports. In this case, the output valid is set to logic 1 to indicate if the data is valid.

- Repeat for the next transaction.

The second recommended flow is for continuous execution of the block. In this mode, the input ports included in the AXI4-Lite slave interface should only be ports which perform configuration. The block will typically run must faster than a CPU. If the block must wait for inputs, the block will spend most of its time waiting:

- Use the interrupt function to determine how you wish the interrupt to operate.
- Load the register values for the block input ports. In the above example this is performed using API functions `XExample_Set_a`, `XExample_Set_a` and `XExample_Set_c_i`.
- Set the auto-start function using API `XExample_EnableAutoRestart`
- Allow the function to execute. The individual port I/O protocols will synchronize the data being processed through the block.
- Address any interrupts which are generated. The output registers could be accessed during this operation but the data may change often.
- Use the API function `XExample_DisableAutoRestart` to prevent any more executions.
- Read the output registers. In the above example this is performed using API functions `XExample_Get_c_o` and `XExample_Set_c_o_vld`.

Controlling Software

The API functions can be used in the software running on the CPU to control the hardware block. An overview of the process is:

- Create an instance of the HW instance
- Look Up the device configuration
- Initialize the Device
- Set the input parameters of the HLS block
- Start the device and read the results

An abstracted versions of this process is shown below. Complete examples of the software control are provided in the Zynq-7000 SoC tutorials.

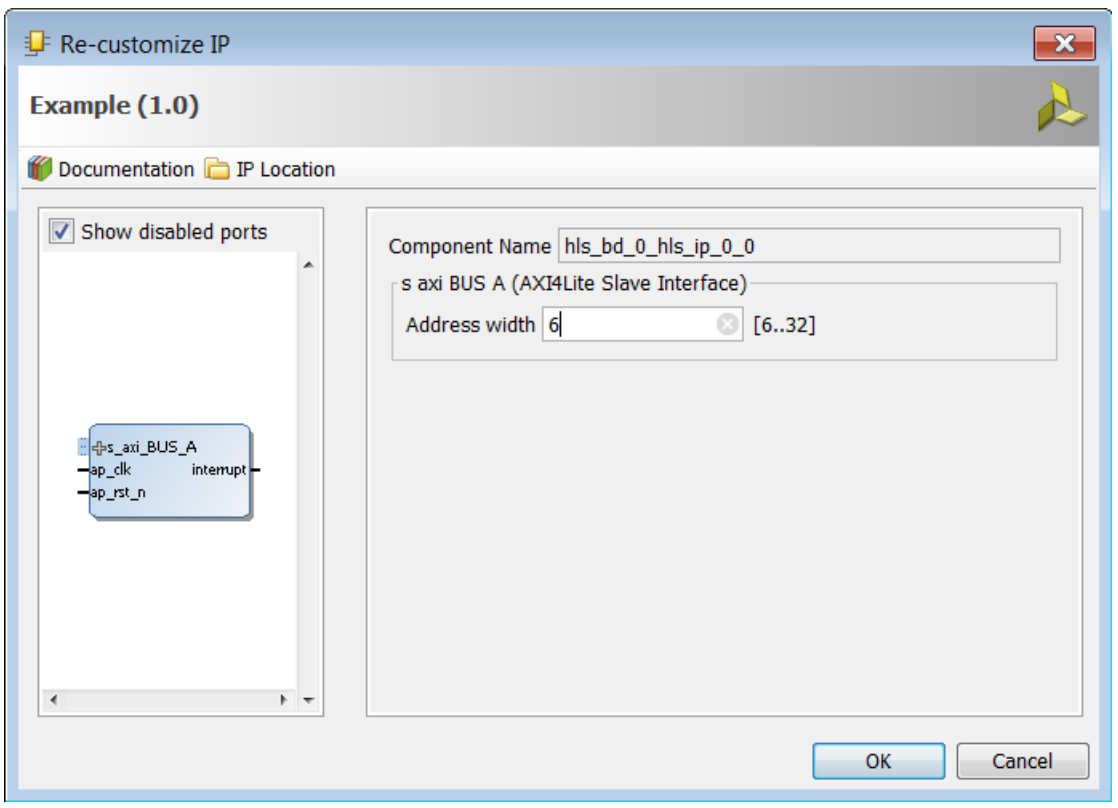
```
#include "xexample.h"    // Device driver for HLS HW block
#include "xparameters.h"

// HLS HW instance
XExample HlsExample;
XExample_Config *ExamplePtr

int main() {
 int res_hw;

// Look Up the device configuration
 ExamplePtr = XExample_LookupConfig(XPAR_XEXAMPLE_0_DEVICE_ID);
 if (!ExamplePtr) {
 print("ERROR: Lookup of accelerator configuration failed.\n\r");
 return XST_FAILURE;
 }

// Initialize the Device
 status = XExample_CfgInitialize(&HlsExample, ExamplePtr);
 if (status != XST_SUCCESS) {
 print("ERROR: Could not initialize accelerator.\n\r");
 exit(-1);
 }

//Set the input parameters of the HLS block
 XExample_Set_a(&HlsExample, 42);
 XExample_Set_b(&HlsExample, 12);
 XExample_Set_c_i(&HlsExample, 1);

// Start the device and read the results
 XExample_Start(&HlsExample);
 do {
res_hw = XExample_Get_c_o(&HlsExample);
} while (XExample_Get_c_o(&HlsExample) == 0); // wait for valid data output
```

Customizing AXI4-Lite Slave Interfaces in IP Integrator

When an HLS RTL design using an AXI4-Lite slave interface is incorporated into a design in Vitis IP Integrator, you can customize the block. From the block diagram in IP Integrator, select the HLS block, right-click with the mouse button and select **Customize Block**.

The address width is by default configured to the minimum required size. Modify this to connect to blocks with address sizes less than 32-bit.

Figure 23: Customizing AXI4-Lite Slave Interfaces in IP Integrator



AXI4 Master Interface

You can use an AXI4 master interface on array or pointer/reference arguments, which Vitis HLS implements in one of the following modes:

- Individual data transfers
- Burst mode data transfers

With individual data transfers, Vitis HLS reads or writes a single element of data for each address. The following example shows a single read and single write operation. In this example, Vitis HLS generates an address on the AXI interface to read a single data value and an address to write a single data value. The interface transfers one data value per address.

```
void bus (int *d) {
 static int acc = 0;

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

With burst mode transfers, Vitis HLS reads or writes data using a single base address followed by multiple sequential data samples, which makes this mode capable of higher data throughput. Burst mode of operation is possible when you use the C `memcpy` function or a pipelined `for` loop.

Note: The C `memcpy` function is only supported for synthesis when used to transfer data to or from a top-level function argument specified with an AXI4 master interface.

The following example shows a copy of burst mode using the `memcpy` function. The top-level function argument `a` is specified as an AXI4 master interface.

```
void example(volatile int *a){

#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return

//Port a is assigned to an AXI4 master interface

 int i;
 int buff[50];

//memcpy creates a burst access to memory
 memcpy(buff,(const int*)a,50*sizeof(int));

 for(i=0; i < 50; i++){
 buff[i] = buff[i] + 100;
 }

 memcpy((int *)a,buff,50*sizeof(int));
}
```

When this example is synthesized, it results in the interface shown in the following figure.

Note: In this figure, the AXI4 interfaces are collapsed.

Figure 24: AXI4 Interface

The following example shows the same code as the preceding example but uses a `for` loop to copy the data out:

```
void example(volatile int *a){

#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return

//Port a is assigned to an AXI4 master interface

 int i;
 int buff[50];

//memcpy creates a burst access to memory
 memcpy(buff,(const int*)a,50*sizeof(int));

 for(i=0; i < 50; i++){
 buff[i] = buff[i] + 100;
 }

 for(i=0; i < 50; i++){
#pragma HLS PIPELINE
 a[i] = buff[i];
 }
}
```

When using a `for` loop to implement burst reads or writes, follow these requirements:

- Pipeline the loop
- Access addresses in increasing order
- Do not place accesses inside a conditional statement
- For nested loops, do not flatten loops, because this inhibits the burst operation

Note: Only one read and one write is allowed in a `for` loop unless the ports are bundled in different AXI ports. The following example shows how to perform two reads in burst mode using different AXI interfaces.

In the following example, Vitis HLS implements the port reads as burst transfers. Port `a` is specified without using the `bundle` option and is implemented in the default AXI interface. Port `b` is specified using a named bundle and is implemented in a separate AXI interface called `d2_port`.

```
void example(volatile int *a, int *b){

#pragma HLS INTERFACE s_axilite port=return
#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE m_axi depth=50 port=b bundle=d2_port


 int i;
 int buff[50];

//copy data in
 for(i=0; i < 50; i++){
#pragma HLS PIPELINE
 buff[i] = a[i] + b[i];
 }
...
 }
```

IMPORTANT: Structs are only supported for the AXI4 master interface if the struct is packed, which is the default in Vitis HLS.

Automatic Port Width Resizing

Vitis HLS provides the ability to automatically size the kernel interface ports to 512 for the burst access, in the Vitis tool flow. This means that the access which can be burst will only be sized to 512 bit-width. For bursting the code should follow certain pre-conditions as discussed in Optimizing Burst Transfers. Vitis HLS controls automatic port width resizing using the following two commands:

- `config_interface -m_axi_max_widen_bitwidth <N>` : Directs the tool to automatically widen bursts on M-AXI interfaces up to the specified bitwidth. The value of <N> must be a power-of-two between 0 and 1024. Note that burst widening requires strong alignment properties (in addition to bursting). The default value for <N> is 512 bits.
- `config_interface -m_axi_alignment_byte_size <N>` : Assume top function pointers that are mapped to M-AXI interfaces are at least aligned to the provided width in bytes (power of two). This can help automatic burst widening. The default alignment is 64 bytes.

For automatic optimization, the kernel needs to satisfy the following preconditions:

- Must be a monotonically increasing order of access (both in terms of the memory location being accessed as well as in time). You cannot access a memory location that is in between two previously accessed memory locations- aka no overlap.
- All the preconditions that apply to burst are also applied to port-width resizing.
- The access pattern from the global memory should be in sequential order, and with the following additional requirements:
    - The sequential accesses need to be on a non-vector type
    - The start of the sequential accesses needs to be aligned to the widen word size
    - The length of the sequential accesses needs to be divisible by the widen factor

The following code example is used in the example calculations that follow:

```
vadd_pipeline:
  for (int i = 0; i < iterations; i++) {
#pragma HLS LOOP_TRIPCOUNT min = c_len/c_n max = c_len/c_n

  // Pipelining loops that access only one variable is the ideal way to
  // increase the global memory bandwidth.
  read_a:
    for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
      result[x] = a[i * N + x];
    }

  read_b:
    for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
      result[x] += b[i * N + x];
    }

  write_c:
    for (int x = 0; x < N; ++x) {
#pragma HLS LOOP_TRIPCOUNT min = c_n max = c_n
#pragma HLS PIPELINE II = 1
      c[i * N + x] = result[x];
    }
  }
}
}
```

Note: The code example above is from the Vitis HLS examples `cpp-kernels/loop_pipeline`.

The width of the automatic optimization for the code above is performed in three steps:

1. First, the tool checks for the number of access patterns in the read_a loop. There is one access during one loop iteration, so the optimization determines the interface bit-width as 32= 32 *1 (bitwidth of the int variable * accesses).
2. The tool tries to reach the default max specified by the `config_interface m_axi_max_widen_bitwidth 512` , using the following expression terms:

   ```
   length = (ceil((loop-bound of index inner loops) * (loop-bound of index - outer loops)) * #(of access-patterns))
   ```

   - In the above code, the outer loop is an imperfect loop so there will not be burst transfers on the outer-loop. Therefore the length will only include the inner-loop. Therefore the formula will be shortened to:

     ```
     length = (ceil((loop-bound of index inner loops)) * #(of access-patterns))
     ```

     or: length = ceil(128) *32 = 4096
3. Finally, is the calculated length a power of 2? If yes then the length will be capped to the width specified by the `m_axi_max_widen_bitwidth` .

There are some pros and cons to using the automatic port width resizing which you should consider when using this feature. This feature improves the read latency from the DDR as the tool is reading a big vector, instead of the data type size. It also adds more resources as it needs to buffer the huge vector and shift the data accordingly to the data path size. However, automatic port width resizing supports only standard C datatypes and does not support non-aggregate type such as ap_int, ap_uint, struct, or array.

Controlling AXI4 Burst Behavior

An optimal AXI4 interface is one in which the design never stalls while waiting to access the bus, and after bus access is granted, the bus never stalls while waiting for the design to read/write. To create the optimal AXI4 interface, the following options are provided in the INTERFACE pragma or directive to specify the behavior of the bursts and optimize the efficiency of the AXI4 interface. Refer to Optimizing Burst Transfers for more information on burst transfers.

Some of these options use internal storage to buffer data and may have an impact on area and resources:

- `latency` : Specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request a number of cycles (latency) before the read or write is expected. If this figure is too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be granted but the bus may stall waiting on the design to start the access.
- `max_read_burst_length` : Specifies the maximum number of data values read during a burst transfer.
- `num_read_outstanding` : Specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size: `num_read_outstanding` * `max_read_burst_length` * `word_size` .
- `max_write_burst_length` : Specifies the maximum number of data values written during a burst transfer.

- `num_write_outstanding` : Specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design, a FIFO of size:
  `num_read_outstanding` * `max_read_burst_length` * `word_size`

The following example can be used to help explain these options:

```
 #pragma HLS interface m_axi port=input offset=slave bundle=gmem0
depth=1024*1024*16/(512/8)
 latency=100
 num_read_outstanding=32
 num_write_outstanding=32
 max_read_burst_length=16
 max_write_burst_length=16
```

The interface is specified as having a latency of 100. Vitis HLS seeks to schedule the request for burst access 100 clock cycles before the design is ready to access the AXI4 bus. To further improve bus efficiency, the options `num_write_outstanding` and `num_read_outstanding` ensure the design contains enough buffering to store up to 32 read and write accesses. This allows the design to continue processing until the bus requests are serviced. Finally, the options `max_read_burst_length` and `max_write_burst_length` ensure the maximum burst size is 16 and that the AXI4 interface does not hold the bus for longer than this.

These options allow the behavior of the AXI4 interface to be optimized for the system in which it will operate. The efficiency of the operation does depend on these values being set accurately.

Creating an AXI4 Interface with 32-bit Address Capability

By default, Vitis HLS implements the AXI4 port with a 64-bit address bus. Optionally, you can implement the AXI4 interface with a 32-bit address bus by disabling the `m_axi_addr64` interface configuration option as follows:

1. Select **Solution** > (and then)**Solution Settings**.
2. In the Solution Settings dialog box, click the **General** category, and **Edit** the existing `config_interface` command, or click **Add** to add one.
3. In the Edit or Add dialog box, select **config_interface**, and disable **m_axi_addr64**.

IMPORTANT: When you select the **m_axi_addr64** option, Vitis HLS implements all AXI4 interfaces in the design with a 32-bit address bus.

Controlling the Address Offset in an AXI4 Interface

By default, the AXI4 master interface starts all read and write operations from address 0x00000000. For example, given the following code, the design reads data from addresses 0x00000000 to 0x000000c7 (50 32-bit words, gives 200 bytes), which represents 50 address values. The design then writes data back to the same addresses.

```
void example(volatile int *a){

#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return bundle=AXILiteS

 int i;
 int buff[50];

 memcpy(buff,(const int*)a,50*sizeof(int));

 for(i=0; i < 50; i++){
 buff[i] = buff[i] + 100;
 }
 memcpy((int *)a,buff,50*sizeof(int));
}
```

To apply an address offset, use the `-offset` option with the INTERFACE directive, and specify one of the following options:

- `off` : Does not apply an offset address. This is the default.
- `direct` : Adds a 32-bit port to the design for applying an address offset.
- `slave` : Adds a 32-bit register inside the AXI4-Lite interface for applying an address offset.

In the final RTL, Vitis HLS applies the address offset directly to any read or write address generated by the AXI4 master interface. This allows the design to access any address location in the system.

If you use the `slave` option in an AXI interface, you must use an AXI4-Lite port on the design interface. Xilinx recommends that you implement the AXI4-Lite interface using the following pragma:

```
#pragma HLS INTERFACE s_axilite port=return
```

In addition, if you use the `slave` option and you used several AXI4-Lite interfaces, you must ensure that the AXI master port offset register is bundled into the correct AXI4-Lite interface. In the following example, port `a` is implemented as an AXI master interface with an offset and AXI4-Lite interfaces called `AXI_Lite_1` and `AXI_Lite_2`:

```
#pragma HLS INTERFACE m_axi port=a depth=50 offset=slave
#pragma HLS INTERFACE s_axilite port=return bundle=AXI_Lite_1
#pragma HLS INTERFACE s_axilite port=b bundle=AXI_Lite_2
```

The following INTERFACE directive is required to ensure that the offset register for port `a` is bundled into the AXI4-Lite interface called `AXI_Lite_1`:

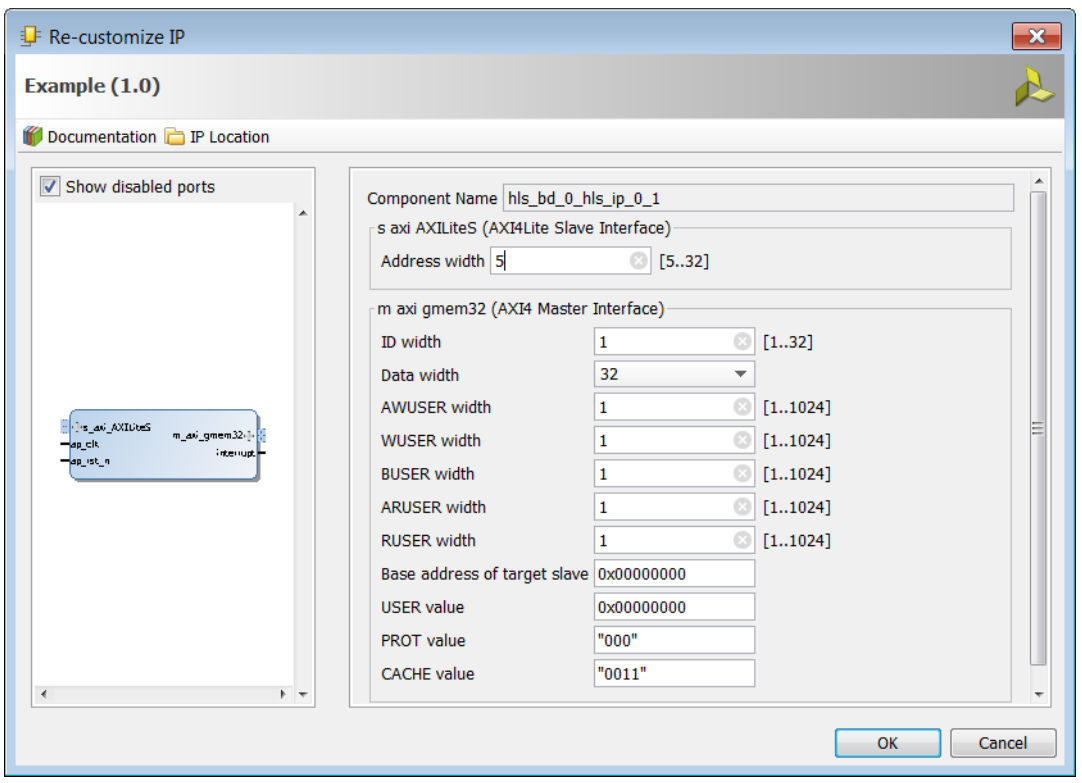
```
#pragma HLS INTERFACE s_axilite port=a bundle=AXI_Lite_1
```

Customizing AXI4 Master Interfaces in IP Integrator

When you incorporate an HLS RTL design that uses an AXI4 master interface into a design in the Vivado IP Integrator, you can customize the block. From the block diagram in IP Integrator, select the HLS block, right-click, and select **Customize Block** to customize any of the settings provided. A complete description of the AXI4 parameters is provided in this link (https://www.xilinx.com/cgi-bin/docs/ipdoc?c=axi_ref_guide;v=latest;d=ug1037-vivado-axi-reference-guide.pdf;a=xAXI4AndAXI4LiteSignals) in the Vivado Design Suite: AXI Reference Guide (UG1037 (https://www.xilinx.com/cgi-bin/docs/ipdoc?c=axi_ref_guide;v=latest;d=ug1037-vivado-axi-reference-guide.pdf)).

The following figure shows the Re-Customize IP dialog box for the design shown below. This design includes an AXI4-Lite port.

Figure 25: Customizing AXI4 Master Interfaces in IP Integrator



Managing Interfaces with SSI Technology Devices

Certain Xilinx devices use stacked silicon interconnect (SSI) technology. In these devices, the total available resources are divided over multiple super logic regions (SLRs). The connections between SLRs use super long line (SSL) routes. SSL routes incur delays costs that are typically greater than standard FPGA routing. To ensure designs operate at maximum performance, use the following guidelines:

- Register all signals that cross between SLRs at both the SLR output and SLR input.
- You do not need to register a signal if it enters or exits an SLR via an I/O buffer.
- Ensure that the logic created by Vitis HLS fits within a single SLR.

Note: When you select an SSI technology device as the target technology, the utilization report includes details on both the SLR usage and the total device usage.

If the logic is contained within a single SLR device, Vitis HLS provides a `register_io` option to the `config_interface` command. This option provides a way to automatically register all block inputs, outputs, or both. This option is only required for scalars. All array ports are automatically registered.

The settings for the `register_io` option are:

- `off` : None of the input or outputs are registered.
- `scalar_in` : All inputs are registered.
- `scalar_out` : All outputs are registered.
- `scalar_all` : All input and outputs are registered.

Note: Using the `register_io` option with block-level floorplanning of the RTL ensures that logic targeted to an SSI technology device executes at the maximum clock rate.

Optimization Techniques in Vitis HLS

This section outlines the various optimization techniques you can use to direct Vitis HLS to produce a micro-architecture that satisfies the desired performance and area goals. Using Vitis HLS, you can apply different optimization directives to the design, including:

- Pipelining tasks, allowing the next execution of the task to begin before the current execution is complete.
- Specifying a target latency for the completion of functions, loops, and regions.
- Specifying a limit on the number of resources used.
- Overriding the inherent or implied dependencies in the code to permit specific operations. For example, if it is acceptable to discard or ignore the initial data values, such as in a video stream, allow a memory read before write if it results in better performance.
- Specifying the I/O protocol to ensure function arguments can be connected to other hardware blocks with the same I/O protocol.

  Note: Vitis HLS automatically determines the I/O protocol used by any sub-functions. You cannot control these ports except to specify whether the port is registered.

The optimizations techniques are presented in the context of how they are typically applied to a design:

- Optimizing for Throughput presents primary optimizations in the order in which they are typically used: pipeline the tasks to improve performance, improve the flow of data between tasks, and optimize structures to improve address issues which may limit performance.
- Optimizing for Latency uses the techniques of latency constraints and the removal of loop transitions to reduce the number of clock cycles required to complete.
- Optimizing for Area focuses on how operations are implemented - controlling the number of operations and how those operations are implemented in hardware - is the principal technique for improving the area.
- Optimizing Logic discusses optimizations affecting the implementation of the RTL.

You can add optimization directives directly into the source code as compiler pragmas using various HLS pragmas, or you can use Tcl `set_directive` commands to apply optimization directives in a Tcl script to be used by a solution during compilation as discussed in Optimizing the HLS Project (optimizinghlsproject.html). The following table lists the optimization directives provided by Vitis HLS as either pragma or Tcl directive.

Table 7. Vitis HLS Optimization Directives

Directive	Description
`ALLOCATION`	Specify a limit for the number of operations, implementations, or functions used. This can force the sharing or hardware resources and may increase latency.
`ARRAY_PARTITION`	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
`ARRAY_RESHAPE`	Reshape an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM.
`BIND_OP`	Define a specific implementation for an operation in the RTL.
`BIND_STORAGE`	Define a specific implementation for a storage element, or memory, in the RTL.
`DATAFLOW`	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to optimize throughput and/or latency.
`DEPENDENCE`	Used to provide additional information that can overcome loop-carried dependencies and allow loops to be pipelined (or pipelined with lower intervals).

Directive	Description
DISAGGREGATE	Break a struct down into its individual elements.
EXPRESSION_BALANCE	Allows automatic expression balancing to be turned off.
INLINE	Inlines a function, removing function hierarchy at this level. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
INTERFACE	Specifies how RTL ports are created from the function description.
LATENCY	Allows a minimum and maximum latency constraint to be specified.
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop with improved latency.
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.
LOOP_TRIPCOUNT	Used for loops which have variables bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.
OCCURRENCE	Used when pipelining functions or loops, to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop.
PIPELINE	Reduces the initiation interval by allowing the overlapped execution of operations within a loop or function.
RESET	This directive is used to add or remove reset on a specific state variable (global or static).
SHARED	Specifies that a global variable, or function argument array is shared among multiple dataflow processes, without the need for synchronization.
STABLE	Indicates that a variable input or output of a dataflow region can be ignored when generating the synchronizations at entry and exit of the dataflow region.
STREAM	Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization. When using hls::stream, the STREAM optimization directive is used to override the configuration of the hls::stream.
TOP	The top-level function for synthesis is specified in the project settings. This directive may be used to specify any function as the top-level for synthesis. This then allows different solutions within the same project to be specified as the top-level function for synthesis without needing to create a new project.
UNROLL	Unroll for-loops to create multiple instances of the loop body and its instructions that can then be scheduled independently.

In addition to the optimization directives, Vitis HLS provides a number of configuration commands that can influence the performance of synthesis results. Details on using configurations commands can be found in Setting Configuration Options (creatingnewvitishlsproject.html#zle1584656100548). The following table reflects some of these commands.

Table 8. Vitis HLS Configurations

GUI Directive	Description
Config Array Partition	Determines how arrays are partitioned, including global arrays and if the partitioning impacts array ports.
Config Compile	Controls synthesis specific optimizations such as the automatic loop pipelining and floating point math optimizations.
Config Dataflow	Specifies the default memory channel and FIFO depth in dataflow optimization.
Config Interface	Controls I/O ports not associated with the top-level function arguments and allows unused ports to be eliminated from the final RTL.
Config Op	Configures the default latency and implementation of specified operations.
Config RTL	Provides control over the output RTL including file and module naming, reset style and FSM encoding.
Config Schedule	Determines the effort level to use during the synthesis scheduling phase and the verbosity of the output messages
Config Storage	Configures the default latency and implementation of specified storage types.

GUI Directive	Description
Config Unroll	Configures the default tripcount threshold for unrolling loops.

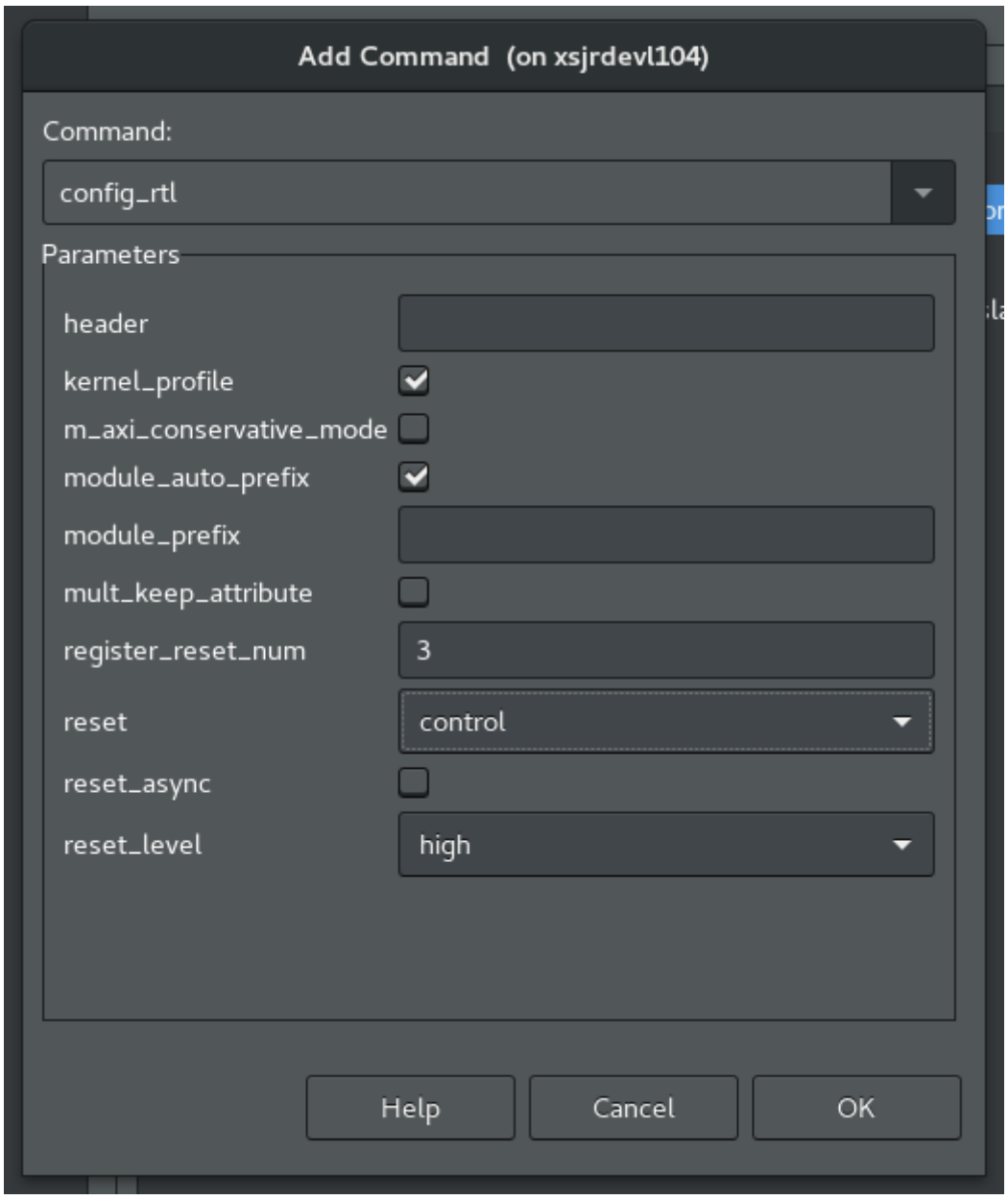
Controlling the Reset Behavior

The reset port is used in an FPGA to return the registers and block RAM connected to the reset port to an initial value any time the reset signal is applied. Typically the most important aspect of RTL configuration is selecting the reset behavior.

Note: When discussing reset behavior it is important to understand the difference between initialization and reset. Refer to Initialization Behavior for more information.

The presence and behavior of the RTL reset port is controlled using the `config_rtl` command, as shown in the following figure. You can access this command by selecting the **Solution** > (and then)**Solution Settings** menu command.

Figure 26: RTL Configurations



The reset settings include the ability to set the polarity of the reset and whether the reset is synchronous or asynchronous but more importantly it controls, through the **reset** option, which registers are reset when the reset signal is applied.

IMPORTANT: When AXI4 interfaces are used on a design the reset polarity is automatically changed to active-Low irrespective of the setting in the `config_rtl` configuration. This is required by the AXI4 standard.

The **reset** option has four settings:

- **none**: No reset is added to the design.
- **control**: This is the default and ensures all control registers are reset. Control registers are those used in state machines and to generate I/O protocol signals. This setting ensures the design can immediately start its operation state.
- **state**: This option adds a reset to control registers (as in the **control** setting) plus any registers or memories derived from static and global variables in the C code. This setting ensures static and global variable initialized in the C code are reset to their initialized value after the reset is applied.
- **all**: This adds a reset to all registers and memories in the design.

Finer grain control over reset is provided through the RESET pragma or directive. Static and global variables can have a reset added through the RESET directive. Variables can also be removed from those being reset by using the RESET directive's `off` option.

IMPORTANT: It is important when using the reset `state` or `all` options to consider the effect on resetting arrays as discussed in Initializing and Resetting Arrays.

Initialization Behavior

In C, variables defined with the static qualifier and those defined in the global scope are initialized to zero, by default. These variables may optionally be assigned a specific initial value. For these initialized variables, the value in the C code is assigned at compile time (at time zero) and never again. In both cases, the initial value is implemented in the RTL.

- During RTL simulation the variables are initialized with the same values as the C code.
- The variables are also initialized in the bitstream used to program the FPGA. When the device powers up, the variables will start in their initialized state.

In the RTL, although the variables start with the same initial value as the C code, there is no way to force the variable to return to this initial state. To restore the initial state, variables must be implemented with a reset signal.

IMPORTANT: Top-level function arguments may be implemented in an AXI4-Lite interface. Since there is no way to provide an initial value in C/C++ for function arguments, these variable cannot be initialized in the RTL as doing so would create an RTL design with different functional behavior from the C/C++ code which would fail to verify during C/RTL co-simulation.

Initializing and Resetting Arrays

Arrays are often defined as static variables, which implies all elements are initialized to zero; and arrays are typically implemented as block RAM. When reset options `state` or `all` are used, it forces all arrays implemented as block RAM to be returned to their initialized state after reset. This may result in two very undesirable conditions in the RTL design:

- Unlike a power-up initialization, an explicit reset requires the RTL design iterate through each address in the block RAM to set the value: this can take many clock cycles if N is large, and requires more area resources to implement the reset.
- A reset is added to every array in the design.

To prevent adding reset logic onto every such block RAM, and incurring the cycle overhead to reset all elements in the RAM, specify the default `control` reset mode and use the RESET directive to identify individual static or global variables to be reset.

Alternatively, you can use the `state` reset mode, and use the RESET directive `off` option to identify individual static or global variables to remove the reset from.

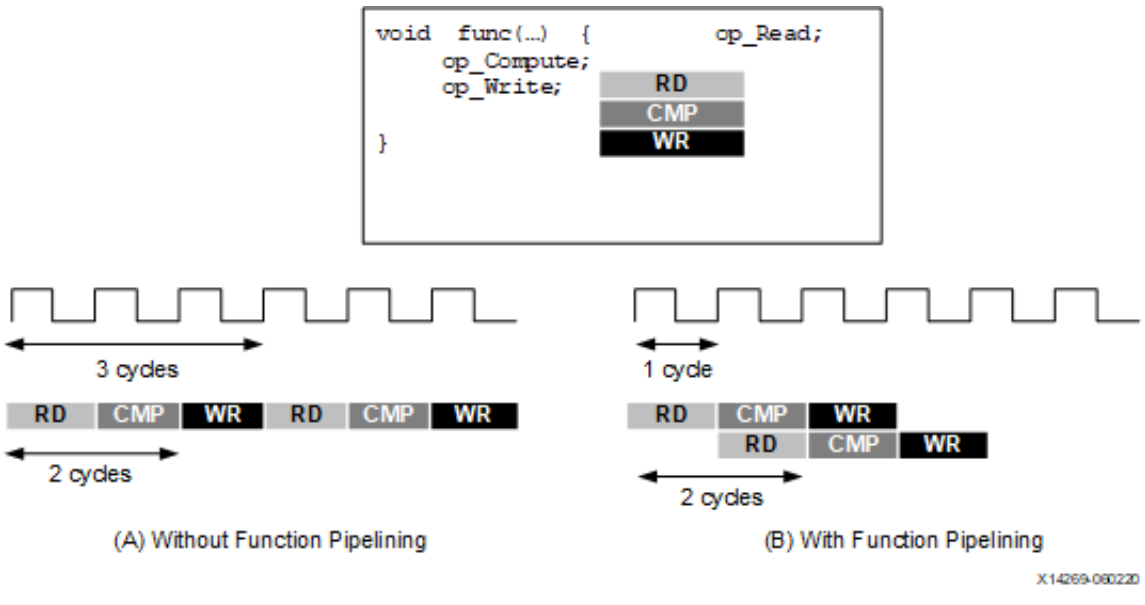
Optimizing for Throughput

Use the following optimizations to improve throughput or reduce the initiation interval.

Function and Loop Pipelining

Pipelining allows operations to happen concurrently: each execution step does not have to complete all operations before it begins the next operation. Pipelining is applied to functions and loops. The throughput improvements in function pipelining are shown in the following figure.

Figure 27: Function Pipelining Behavior

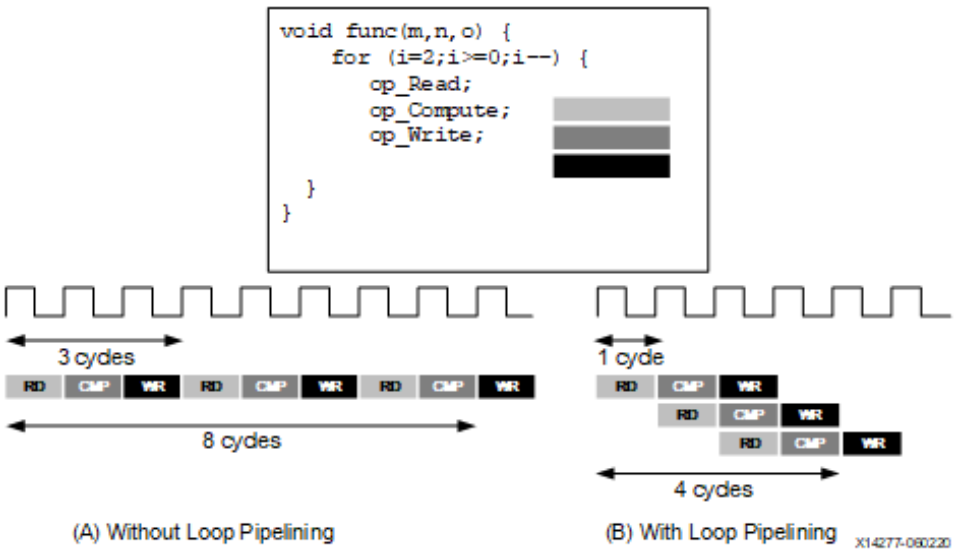


Without pipelining, the function in the above example reads an input every 3 clock cycles and outputs a value after 2 clock cycles. The function has an initiation interval (II) of 3 and a latency of 3. With pipelining, for this example, a new input is read every cycle (II=1) with no change to the output latency.

Loop pipelining allows the operations in a loop to be implemented in an overlapping manner. In the following figure, (A) shows the default sequential operation where there are 3 clock cycles between each input read (II=3), and it requires 8 clock cycles before the last output write is performed.

In the pipelined version of the loop shown in (B), a new input sample is read every cycle (II=1) and the final output is written after only 4 clock cycles: substantially improving both the II and latency while using the same hardware resources.

Figure 28: Loop Pipelining



Functions or loops are pipelined using the PIPELINE directive. The directive is specified in the region that constitutes the function or loop body. The initiation interval defaults to 1 if not specified but may be explicitly specified.

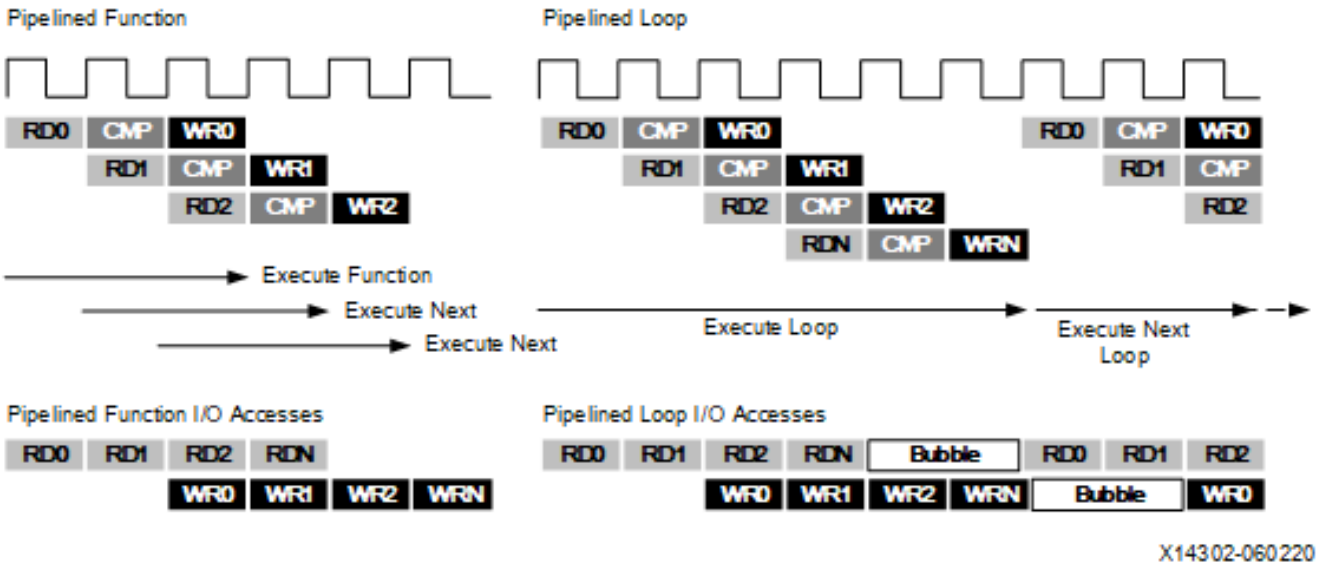
Pipelining is applied only to the specified region and not to the hierarchy below. However, all loops in the hierarchy below are automatically unrolled. Any sub-functions in the hierarchy below the specified function must be pipelined individually. If the sub-functions are pipelined, the pipelined functions above it can take advantage of the pipeline performance. Conversely, any sub-function below the pipelined top-level function that is not pipelined might be the limiting factor in the performance of the pipeline.

There is a difference in how pipelined functions and loops behave.

- In the case of functions, the pipeline runs forever and never ends.
- In the case of loops, the pipeline executes until all iterations of the loop are completed.

This difference in behavior is summarized in the following figure.

Figure 29: Function and Loop Pipelining Behavior



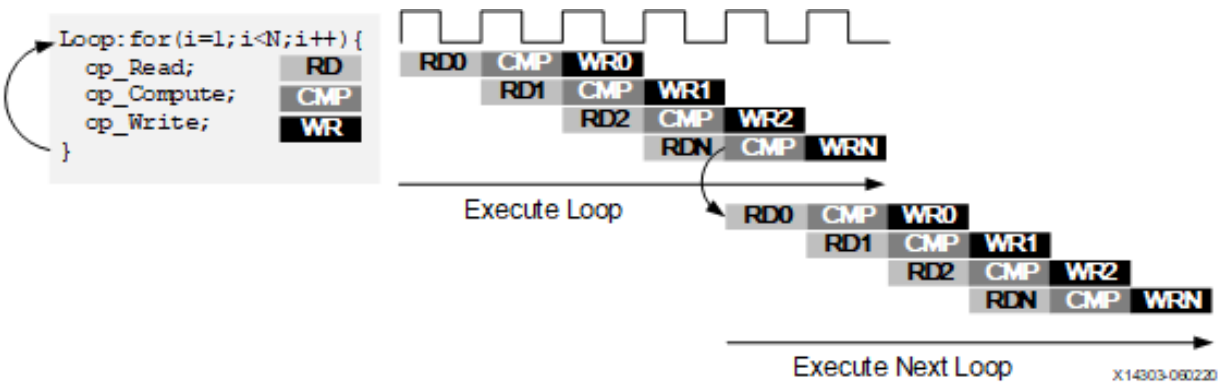
The difference in behavior impacts how inputs and outputs to the pipeline are processed. As seen in the figure above, a pipelined function will continuously read new inputs and write new outputs. By contrast, because a loop must first finish all operations in the loop before starting the next loop, a pipelined loop causes a "bubble" in the data stream; that is, a point when no new inputs are read as the loop completes the execution of the final iterations, and a point when no new outputs are written as the loop starts new loop iterations.

Rewinding Pipelined Loops for Performance

To avoid issues shown in the previous figure, the PIPELINE pragma has an optional command `rewind`. This command enables the overlap of the iterations of successive calls to the `rewind` loop, when this loop is the outermost construct of the top function or of a dataflow process (and the dataflow region is called multiple times).

The following figure shows the operation when the `rewind` option is used when pipelining a loop. At the end of the loop iteration count, the loop starts to re-execute. While it generally re-executes immediately, a delay is possible and is shown and described in the GUI.

Figure 30: Loop Pipelining with Rewind Option

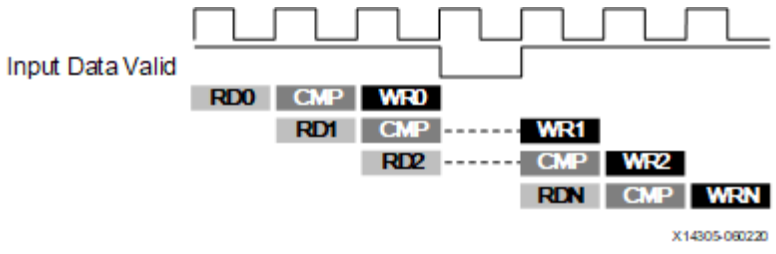


Note: If a loop is used around a DATAFLOW region, Vitis HLS automatically implements it to allow successive iterations to overlap. See Exploiting Task Level Parallelism: Dataflow Optimization for more information.

Flushing Pipelines

Pipelines continue to execute as long as data is available at the input of the pipeline. If there is no data available to process, the pipeline will stall. This is shown in the following figure, where the input data valid signal goes low to indicate there is no more data. Once there is new data available to process, the pipeline will continue operation.

Figure 31: Loop Pipelining with Stall



In some cases, it is desirable to have a pipeline that can be "emptied" or "flushed." The `flush` option is provided to perform this. When a pipeline is "flushed" the pipeline stops reading new inputs when none are available (as determined by a data valid signal at the start of the pipeline) but continues processing, shutting down each successive pipeline stage, until the final input has been processed through to the output of the pipeline.

The default style of pipelining implemented by Vitis HLS is defined by the `config_compile -pipeline_style` command. You can specify stalling pipelines (stp), or free-running flushing pipelines (frp) to be used throughout the design. You can also define a third type of flushable pipeline (flp) with the PIPELINE pragma or directive, using the `enable_flush` option.

This option applies to the specific scope of the pragma or directive only, and does not change the global default assigned by `config_compile`.

The three types of pipelines available in the tool are summarized in the following table:

Name	**Stalled Pipeline (default)**	**Free running/ Flushable Pipeline**	**Flushable pipeline**
Global Setting	config_compile -pipeline_style stp (default)	config_compile -pipeline_style frp	N/A
Pragma/Directive	#HLS pragma pipeline	N/A	#HLS pragma pipeline enable_flush

Advantages	• Default pipeline. No usage constraints.
	• Typically the lowest overall resource usage.
		• Better timing due to
		  ◦ Less fanout
		  ◦ Simpler pipeline control logic
		• Flushable
			• Flushable

Disadvantages	• Not flushable, hence it may:
	  ◦ Cause more deadlocks in dataflow
	  ◦ Prevent already computed outputs from being delivered, if the inputs to the next iterations are missing
	• Timing issues due to high fanout on pipeline controls
		• Moderate resource increase due to FIFOs added on outputs
		• Usage constraints:
		  ◦ Mainly used for dataflow internal processes
		  ◦ MAXI not supported
			• May have larger II
			• Greater resource usage due to less sharing(II>1)
			• Usage constraints:
			  ◦ Function pipeline only

Use cases	• When there is no timing issue due to high fanout on pipeline control • When flushable is not required (such as no performance or deadlock issue due to stall)	• When you need better timing due to fanout to register enables from pipeline control • When flushable is required for better performance or avoiding deadlock	• When flushable is required for better performance or avoiding deadlock

Automatic Loop Pipelining

The `config_compile` configuration enables loops to be pipelined automatically based on the iteration count. This configuration is accessed through the menu **Solution** > (and then)**Solution Setting** > (and then)**General** > (and then)**Add** > (and then)**config_compile**.

The `pipeline_loops` option sets the iteration limit. All loops with an iteration count below this limit are automatically pipelined. The default is 0: no automatic loop pipelining is performed.

Given the following example code:

```
for (y = 0; y < 480; y++) {
  for (x = 0; x < 640; x++) {
    for (i = 0; i < 5; i++) {
      // do something 5 times
 …      ...
    }
  }
}
```

If the `pipeline_loops` option is set to 6, the innermost `for` loop in the above code snippet will be automatically pipelined. This is equivalent to the following code snippet:

```
for (y = 0; y < 480; y++) {
  for (x = 0; x < 640; x++) {
    for (i = 0; i < 5; i++) {
#pragma HLS PIPELINE II=1
      // do something 5 times
 …    ...
    }
  }
}
```

If there are loops in the design for which you do not want to use automatic pipelining, apply the PIPELINE directive with the `off` option to that loop. The `off` option prevents automatic loop pipelining.

IMPORTANT: Vitis HLS applies the `config_compile pipeline_loops` option after performing all user-specified directives. For example, if Vitis HLS applies a user-specified UNROLL directive to a loop, the loop is first unrolled, and automatic loop pipelining cannot be applied.

Addressing Failure to Pipeline

When a function is pipelined, all loops in the hierarchy below are automatically unrolled. This is a requirement for pipelining to proceed. If a loop has variable bounds it cannot be unrolled. This will prevent the function from being pipelined.

Static Variables

Static variables are used to keep data between loop iterations, often resulting in registers in the final implementation. If this is encountered in pipelined functions, Vitis HLS might not be able to optimize the design sufficiently, which would result in initiation intervals longer than required.

The following is a typical example of this situation:

```
function_foo()
{
        static bool change = 0
        if (condition_xyz){
                change = x; // store
        }
        y = change; // load
}
```

If Vitis HLS cannot optimize this code, the stored operation requires a cycle and the load operation requires an additional cycle. If this function is part of a pipeline, the pipeline has to be implemented with a minimum initiation interval of 2 as the static change variable creates a loop-carried dependency.

One way the user can avoid this is to rewrite the code, as shown in the following example. It ensures that only a read or a write operation is present in each iteration of the loop, which enables the design to be scheduled with II=1.

```
function_readstream()
{
        static bool change = 0
        bool change_temp = 0;
        if (condition_xyz)
        {
                change = x; // store
                change_temp = x;
        }
        else
        {
        change_temp = change; // load
        }
        y = change_temp;
}
```

Partitioning Arrays to Improve Pipelining

A common issue when pipelining functions is the following message:

```
INFO: [SCHED 204-61] Pipelining loop 'SUM_LOOP'.
WARNING: [SCHED 204-69] Unable to schedule 'load' operation ('mem_load_2',
bottleneck.c:62) on array 'mem' due to limited memory ports.
WARNING: [SCHED 204-69] The resource limit of core:RAM:mem:p0 is 1, current
assignments:
WARNING: [SCHED 204-69]     'load' operation ('mem_load', bottleneck.c:62) on array
'mem',
WARNING: [SCHED 204-69] The resource limit of core:RAM:mem:p1 is 1, current
assignments:
WARNING: [SCHED 204-69]     'load' operation ('mem_load_1', bottleneck.c:62) on array
'mem',
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

In this example, Vitis HLS states it cannot reach the specified initiation interval (II) of 1 because it cannot schedule a `load` (read) operation (`mem_load_2`) onto the memory because of limited memory ports. The above message notes that the resource limit for " `core:RAM:mem:p0 is 1` " which is used by the operation `mem_load` on line 62. The 2nd port of the block RAM also only has 1 resource, which is also used by operation `mem_load_1` . Due to this memory port contention, Vitis HLS reports a final II of 2 instead of the desired 1.

This issue is typically caused by arrays. Arrays are implemented as block RAM which only has a maximum of two data ports. This can limit the throughput of a read/write (or load/store) intensive algorithm. The bandwidth can be improved by splitting the array (a single block RAM resource) into multiple smaller arrays (multiple block RAMs), effectively increasing the number of ports.

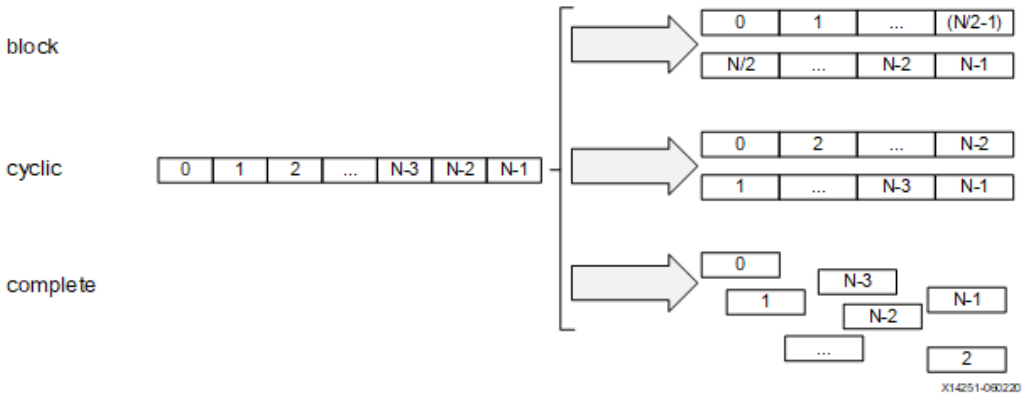
Arrays are partitioned using the ARRAY_PARTITION directive. Vitis HLS provides three types of array partitioning, as shown in the following figure. The three styles of partitioning are:

`block`
: The original array is split into equally sized blocks of consecutive elements of the original array.

`cyclic`
: The original array is split into equally sized blocks interleaving the elements of the original array.

`complete`
: The default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.

Figure 32: Array Partitioning



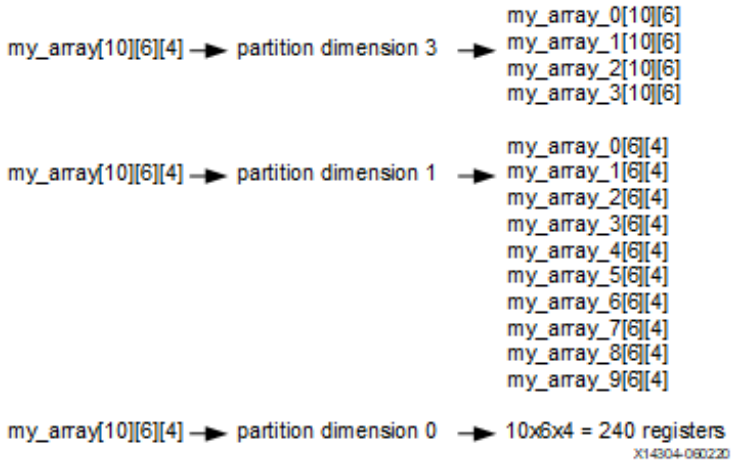
For `block` and `cyclic` partitioning the `factor` option specifies the number of arrays that are created. In the preceding figure, a factor of 2 is used, that is, the array is divided into two smaller arrays. If the number of elements in the array is not an integer multiple of the factor, the final array has fewer elements.

When partitioning multi-dimensional arrays, the `dimension` option is used to specify which dimension is partitioned. The following figure shows how the `dimension` option is used to partition the following example code:

```
void foo (...) {
 int  my_array[10][6][4];
   ...
}
```

The examples in the figure demonstrate how partitioning `dimension` 3 results in 4 separate arrays and partitioning `dimension` 1 results in 10 separate arrays. If zero is specified as the `dimension` , all dimensions are partitioned.

Figure 33: Partitioning Array Dimensions



Automatic Array Partitioning

The `config_array_partition` configuration determines how arrays are automatically partitioned based on the number of elements. This configuration is accessed through the menu **Solution** > (and then)**Solution Settings** > (and then)**General** > (and then)**Add** > (and then)**config_array_partition**.

The partition thresholds can be adjusted and partitioning can be fully automated with the `throughput_driven` option. When the `throughput_driven` option is selected, Vitis HLS automatically partitions arrays to achieve the specified throughput.

Dependencies with Vitis HLS

Vitis HLS constructs a hardware datapath that corresponds to the C source code.

When there is no pipeline directive, the execution is sequential so there are no dependencies to take into account. But when the design has been pipelined, the tool needs to deal with the same dependencies as found in processor architectures for the hardware that Vitis HLS generates.

Typical cases of data dependencies or memory dependencies are when a read or a write occurs after a previous read or write.

- A read-after-write (RAW), also called a true dependency, is when an instruction (and data it reads/uses) depends on the result of a previous operation.
    - I1: t = a * b;
    - I2: c = t + 1;

  The read in statement I2 depends on the write of t in statement I1. If the instructions are reordered, it uses the previous value of t.

- A write-after-read (WAR), also called an anti-dependence, is when an instruction cannot update a register or memory (by a write) before a previous instruction has read the data.
    - I1: b = t + a;
    - I2: t = 3;

  The write in statement I2 cannot execute before statement I1, otherwise the result of b is invalid.

- A write-after-write (WAW) is a dependence when a register or memory must be written in specific order otherwise other instructions might be corrupted.
    - I1: t = a * b;
    - I2: c = t + 1;
    - I3: t = 1;

  The write in statement I3 must happen after the write in statement I1. Otherwise, the statement I2 result is incorrect.

- A read-after-read has no dependency as instructions can be freely reordered if the variable is not declared as volatile. If it is, then the order of instructions has to be maintained.

For example, when a pipeline is generated, the tool needs to take care that a register or memory location read at a later stage has not been modified by a previous write. This is a true dependency or read-after-write (RAW) dependency. A specific example is:

```
int top(int a, int b) {
 int t,c;
I1: t = a * b;
I2: c = t + 1;
 return c;
}
```

Statement `I2` cannot be evaluated before statement `I1` completes because there is a dependency on variable `t`. In hardware, if the multiplication takes 3 clock cycles, then `I2` is delayed for that amount of time. If the above function is pipelined, then VHLS detects this as a true dependency and schedules the operations accordingly. It uses data forwarding optimization to remove the RAW dependency, so that the function can operate at II =1.

Memory dependencies arise when the example applies to an array and not just variables.

```
int top(int a) {
 int r=1,rnext,m,i,out;
 static int mem[256];
L1: for(i=0;i<=254;i++) {
#pragma HLS PIPELINE II=1
I1:     m = r * a; mem[i+1] = m;    // line 7
I2:     rnext = mem[i]; r = rnext; // line 8
 }
 return r;
}
```

In the above example, scheduling of loop `L1` leads to a scheduling warning message:

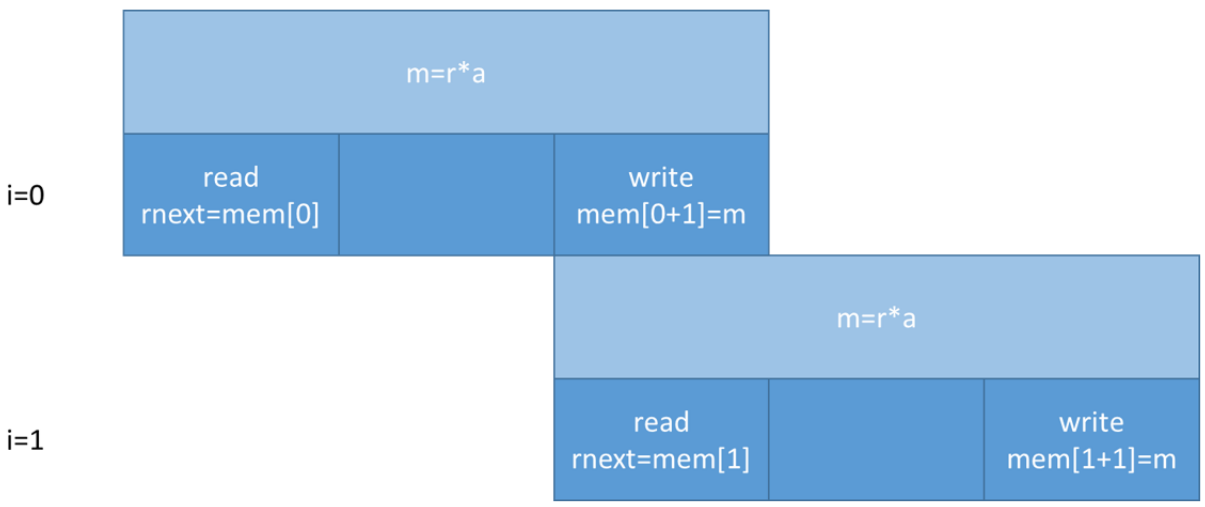
```
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint (II = 1,
distance = 1)
 between 'store' operation (top.cpp:7) of variable 'm', top.cpp:7 on array 'mem' and
'load' operation ('rnext', top.cpp:8) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

There are no issues within the same iteration of the loop as you write an index and read another one. The two instructions could execute at the same time, concurrently. However, observe the read and writes over a few iterations:

```
// Iteration for i=0
I1:     m = r * a; mem[1] = m;      // line 7
I2:     rnext = mem[0]; r = rnext; // line 8
// Iteration for i=1
I1:     m = r * a; mem[2] = m;      // line 7
I2:     rnext = mem[1]; r = rnext; // line 8
// Iteration for i=2
I1:     m = r * a; mem[3] = m;      // line 7
I2:     rnext = mem[2]; r = rnext; // line 8
```

When considering two successive iterations, the multiplication result `m` (with a latency = 2) from statement `I1` is written to a location that is read by statement `I2` of the next iteration of the loop into `rnext`. In this situation, there is a RAW dependence as the next loop iteration cannot start reading `mem[i]` before the previous computation's write completes.

Figure 34: Dependency Example



Note that if the clock frequency is increased, then the multiplier needs more pipeline stages and increased latency. This will force II to increase as well.

Consider the following code, where the operations have been swapped, changing the functionality.

```
int top(int a) {
 int r,m,i;
 static int mem[256];
L1: for(i=0;i<=254;i++) {
#pragma HLS PIPELINE II=1
I1:     r = mem[i];             // line 7
I2:     m = r * a , mem[i+1]=m; // line 8
 }
 return r;
}
```

The scheduling warning is:

```
INFO: [SCHED 204-61] Pipelining loop 'L1'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint (II = 1,
distance = 1)
 between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array 'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint (II = 2,
distance = 1)
 between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array 'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
WARNING: [SCHED 204-68] Unable to enforce a carried dependency constraint (II = 3,
distance = 1)
 between 'store' operation (top.cpp:8) of variable 'm', top.cpp:8 on array 'mem'
and 'load' operation ('r', top.cpp:7) on array 'mem'.
INFO: [SCHED 204-61] Pipelining result: Target II: 1, Final II: 4, Depth: 4.
```

Observe the continued read and writes over a few iterations:

```
Iteration with i=0
I1:     r = mem[0];           // line 7
I2:     m = r * a , mem[1]=m; // line 8
Iteration with i=1
I1:     r = mem[1];           // line 7
I2:     m = r * a , mem[2]=m; // line 8
Iteration with i=2
I1:     r = mem[2];           // line 7
I2:     m = r * a , mem[3]=m; // line 8
```

A longer II is needed because the RAW dependence is via reading `r` from `mem[i]`, performing the multiplication, and writing to `mem[i+1]`.

Removing False Dependencies to Improve Loop Pipelining

False dependencies are dependencies that arise when the compiler is too conservative. These dependencies do not exist in the real code, but cannot be determined by the compiler. These dependencies can prevent loop pipelining.

The following example illustrates false dependencies. In this example, the read and write accesses are to two different addresses in the same loop iteration. Both of these addresses are dependent on the input data, and can point to any individual element of the `hist` array. Because of this, Vitis HLS assumes that both of these accesses can access the same location. As a result, it schedules the read and write operations to the array in alternating cycles, resulting in a loop II of 2. However, the code shows that `hist[old]` and `hist[val]` can never access the same location because they are in the else branch of the conditional `if(old == val)`.

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) f
  int acc = 0;
  int i, val;
  int old = in[0];
  for(i = 0; i < INPUT SIZE; i++)
  {
    #pragma HLS PIPELINE II=1
    val = in[i];
    if(old == val)
    {
        acc = acc + 1;
    }
    else
    {
        hist[old] = acc;
        acc = hist[val] + 1;
    }

    old = val;
}

hist[old] = acc;
```

To overcome this deficiency, you can use the DEPENDENCE directive to provide Vitis HLS with additional information about the dependencies.

```
void histogram(int in[INPUT SIZE], int hist[VALUE SIZE]) {
  int acc = 0;
  int i, val;
  int old = in[0];
  #pragma HLS DEPENDENCE variable=hist intra RAW false
  for(i = 0; i < INPUT SIZE; i++)
  {
    #pragma HLS PIPELINE II=1
    val = in[i];
    if(old == val)
    {
        acc = acc + 1;
    }
    else
    {
        hist[old] = acc;
        acc = hist[val] + 1;
    }

    old = val;
  }

  hist[old] = acc;
```

Note: Specifying a FALSE dependency, when in fact the dependency is not FALSE, can result in incorrect hardware. Be sure dependencies are correct (TRUE or FALSE) before specifying them.

When specifying dependencies there are two main types:

Inter

Specifies the dependency is between different iterations of the same loop.

If this is specified as FALSE it allows Vitis HLS to perform operations in parallel if the pipelined or loop is unrolled or partially unrolled and prevents such concurrent operation when specified as TRUE.

Intra

Specifies dependence within the same iteration of a loop, for example an array being accessed at the start and end of the same iteration.

When intra dependencies are specified as FALSE, Vitis HLS may move operations freely within the loop, increasing their mobility and potentially improving performance or area. When the dependency is specified as TRUE, the operations must be performed in the order specified.

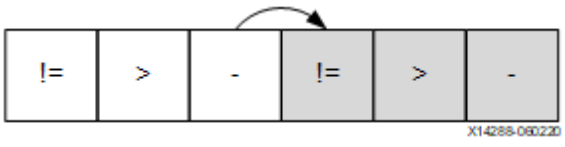
Scalar Dependencies

Some scalar dependencies are much harder to resolve and often require changes to the source code. A scalar data dependency could look like the following:

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

The next iteration of this loop cannot start until the current iteration has calculated the updated the values of `a` and `b`, as shown in the following figure.

Figure 35: Scalar Dependency



If the result of the previous loop iteration must be available before the current iteration can begin, loop pipelining is not possible. If Vitis HLS cannot pipeline with the specified initiation interval, it increases the initiation internal. If it cannot pipeline at all, as shown by the above example, it halts pipelining and proceeds to output a non-pipelined design.

Optimal Loop Unrolling to Improve Pipelining

By default, loops are kept rolled in Vitis HLS. These rolled loops generate a hardware resource which is used by each iteration of the loop. While this creates a resource efficient block, it can sometimes be a performance bottleneck.

Vitis HLS provides the ability to unroll or partially unroll for-loops using the UNROLL directive.

The following figure shows both the advantages of loop unrolling and the implications that must be considered when unrolling loops. This example assumes the arrays `a[i], b[i], and c[i]` are mapped to block RAMs. This example shows how easy it is to create many different implementations by the simple application of loop unrolling.

Figure 36: Loop Unrolling Details

Rolled Loop

When the loop is rolled, each iteration is performed in separate clock cycles. This implementation takes four clock cycles, only requires one multiplier and each block RAM can be a single-port block RAM.

Partially Unrolled Loop

In this example, the loop is partially unrolled by a factor of 2. This implementation required two multipliers and dual-port RAMs to support two reads or writes to each RAM in the same clock cycle. This implementation does however only take 2 clock cycles to complete: half the initiation interval and half the latency of the rolled loop version.

Unrolled loop

In the fully unrolled version all loop operation can be performed in a single clock cycle. This implementation however requires four multipliers. More importantly, this implementation requires the ability to perform 4 reads and 4 write operations in the same clock cycle. Because a block RAM only has a maximum of two ports, this implementation requires the arrays be partitioned.

To perform loop unrolling, you can apply the UNROLL directives to individual loops in the design. Alternatively, you can apply the UNROLL directive to a function, which unrolls all loops within the scope of the function.

If a loop is completely unrolled, all operations will be performed in parallel if data dependencies and resources allow. If operations in one iteration of the loop require the result from a previous iteration, they cannot execute in parallel but will execute as soon as the data is available. A completely unrolled and fully optimized loop will generally involve multiple copies of the logic in the loop body.

The following example code demonstrates how loop unrolling can be used to create an optimized design. In this example, the data is stored in the arrays as interleaved channels. If the loop is pipelined with II=1, each channel is only read and written every 8th block cycle.

```
// Array Order :  0  1  2  3  4  5  6  7  8     9     10    etc. 16       etc...
// Sample Order:  A0 B0 C0 D0 E0 F0 G0 H0 A1    B1    C2    etc. A2       etc...
// Output Order:  A0 B0 C0 D0 E0 F0 G0 H0 A0+A1 B0+B1 C0+C2 etc. A0+A1+A2 etc...

#define CHANNELS 8
#define SAMPLES  400
#define N CHANNELS * SAMPLES

void foo (dout_t d_out[N], din_t d_in[N]) {
 int i, rem;

  // Store accumulated data
 static dacc_t acc[CHANNELS];

 // Accumulate each channel
 For_Loop: for (i=0;i<N;i++) {
 rem=i%CHANNELS;
 acc[rem] = acc[rem] + d_in[i];
 d_out[i] = acc[rem];
 }
}
```

Partially unrolling the loop by a `factor` of 8 will allow each of the channels (every 8th sample) to be processed in parallel (if the input and output arrays are also partitioned in a `cyclic` manner to allow multiple accesses per clock cycle). If the loop is also pipelined with the `rewind` option, this design will continuously process all 8 channels in parallel if called in a pipelined fashion (i.e., either at the top, or within a dataflow region).

```
void foo (dout_t d_out[N], din_t d_in[N]) {
#pragma HLS ARRAY_PARTITION variable=d_i cyclic factor=8 dim=1 partition
#pragma HLS ARRAY_PARTITION variable=d_o cyclic factor=8 dim=1 partition

 int i, rem;

  // Store accumulated data
 static dacc_t acc[CHANNELS];

 // Accumulate each channel
 For_Loop: for (i=0;i<N;i++) {
#pragma HLS PIPELINE rewind
#pragma HLS UNROLL factor=8

 rem=i%CHANNELS;
 acc[rem] = acc[rem] + d_in[i];
 d_out[i] = acc[rem];
 }
}
```

Partial loop unrolling does not require the unroll factor to be an integer multiple of the maximum iteration count. Vitis HLS adds an exit checks to ensure partially unrolled loops are functionally identical to the original loop. For example, given the following code:

```
for(int i = 0; i < N; i++) {
  a[i] = b[i] + c[i];
}
```

Loop unrolling by a factor of 2 effectively transforms the code to look like the following example where the `break` construct is used to ensure the functionality remains the same:

```
for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  if (i+1 >= N) break;
  a[i+1] = b[i+1] + c[i+1];
}
```

Because N is a variable, Vitis HLS may not be able to determine its maximum value (it could be driven from an input port). If the unrolling factor, which is 2 in this case, is an integer factor of the maximum iteration count N, the `skip_exit_check` option removes the exit check and associated logic. The effect of unrolling can now be represented as:

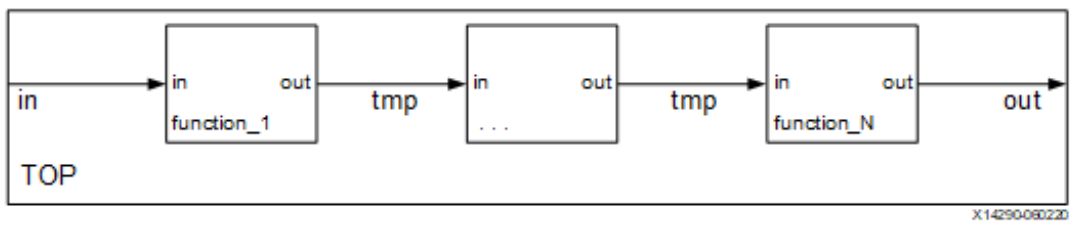
```
for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

This helps minimize the area and simplify the control logic.

Exploiting Task Level Parallelism: Dataflow Optimization

The dataflow optimization is useful on a set of sequential tasks (e.g., functions and/or loops), as shown in the following figure.

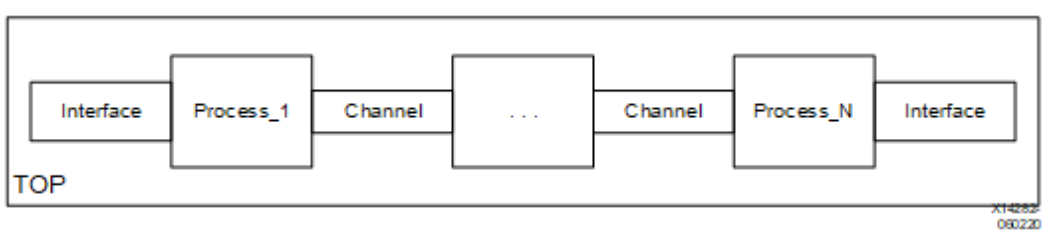
Figure 37: Sequential Functional Description



The above figure shows a specific case of a chain of three tasks, but the communication structure can be more complex than shown.

Using this series of sequential tasks, dataflow optimization creates an architecture of concurrent processes, as shown below. Dataflow optimization is a powerful method for improving design throughput and latency.

Figure 38: Parallel Process Architecture

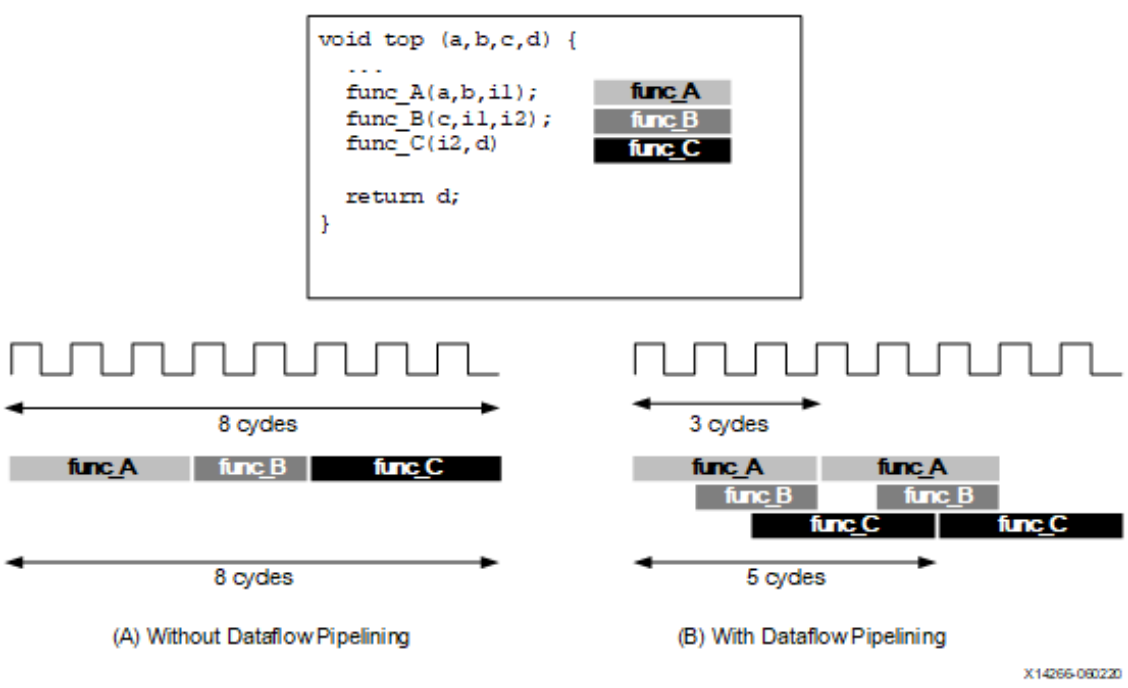


The following figure shows how dataflow optimization allows the execution of tasks to overlap, increasing the overall throughput of the design and reducing latency.

In the following figure and example, (A) represents the case without the dataflow optimization. The implementation requires 8 cycles before a new input can be processed by `func_A` and 8 cycles before an output is written by `func_C`.

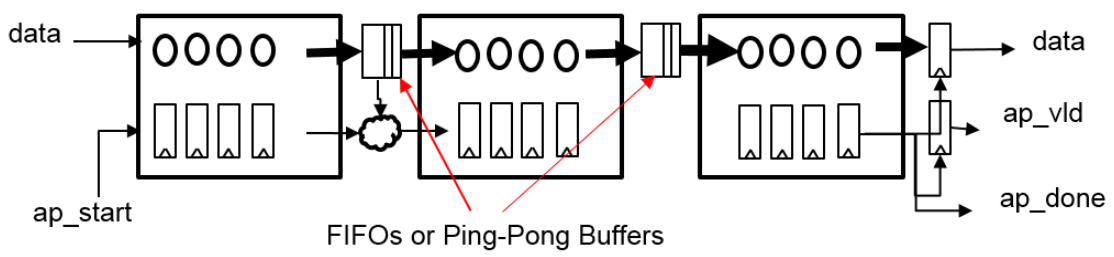
For the same example, (B) represents the case when the dataflow optimization is applied. `func_A` can begin processing a new input every 3 clock cycles (lower initiation interval) and it now only requires 5 clocks to output a final value (shorter latency).

Figure 39: Dataflow Optimization



This type of parallelism cannot be achieved without incurring some overhead in hardware. When a particular region, such as a function body or a loop body, is identified as a region to apply the dataflow optimization, Vitis HLS analyzes the function or loop body and creates individual channels that model the dataflow to store the results of each task in the dataflow region. These channels can be simple FIFOs for scalar variables, or ping-pong (PIPO) buffers for non-scalar variables like arrays. Each of these channels also contain signals to indicate when the FIFO or the ping-pong buffer is full or empty. These signals represent a handshaking interface that is completely data driven. By having individual FIFOs and/or ping-pong buffers, Vitis HLS frees each task to execute at its own pace and the throughput is only limited by availability of the input and output buffers. This allows for better interleaving of task execution than a normal pipelined implementation but does so at the cost of additional FIFO or block RAM registers for the ping-pong buffer. The preceding figure illustrates the structure that is realized for the dataflow region for the same example in the following figure.

Figure 40: Structure Created During Dataflow Optimization



Dataflow optimization potentially improves performance over a statically pipelined solution. It replaces the strict, centrally-controlled pipeline stall philosophy with more flexible and distributed handshaking architecture using FIFOs and/or ping-pong buffers (PIPOs). The replacement of the centralized control structure with a distributed one also benefits the fanout of control signals, for example register enables, which is distributed among the control structures of individual processes.

Dataflow optimization is not limited to a chain of processes, but can be used on any directed acyclic graph (DAG) structure. It can produce two different forms of overlapping: within an iteration if processes are connected with FIFOs, and across different iterations via PIPOs and FIFOs.

Canonical Forms

Vitis HLS transforms the region to apply the DATAFLOW optimization. Xilinx recommends writing the code inside this region (referred to as the *canonical region*) using canonical forms. There are two main canonical forms for the dataflow optimization:

1. The canonical form for a function where functions are not inlined.

```
void dataflow(Input0, Input1, Output0, Output1)
{
 #pragma HLS dataflow
 UserDataType C0, C1, C2;
 func1(read Input0, read Input1, write C0, write C1);
 func2(read C0, read C1, write C2);
 func3(read C2, write Output0, write Output1);
}
```

2. Dataflow inside a loop body.

   For the for loop (where no function inside is inlined), the integral loop variable should have:

   a. Initial value declared in the loop header and set to 0.
   b. The loop condition is a positive numerical constant or constant function argument.
   c. Increment by 1.
   d. Dataflow pragma needs to be inside the loop.

```
void dataflow(Input0, Input1, Output0, Output1)
{
 for (int i = 0; i < N; i++)
 {
 #pragma HLS dataflow
 UserDataType C0, C1, C2;
 func1(read Input0, read Input1, write C0, write C1);
 func2(read C0, read C0, read C1, write C2);
 func3(read C2, write Output0, write Output1);
 }
}
```

Canonical Body

Inside the canonical region, the canonical body should follow these guidelines:

1. Use a local, non-static scalar or array/pointer variable, or local static stream variable. A local variable is declared inside the function body (for dataflow in a function) or loop body (for dataflow inside a loop).
2. A sequence of function calls that pass data forward (with no feedback), from a function to one that is lexically later, under the following conditions:
   a. Variables (except scalar) can have only one reading process and one writing process.
   b. Use write before read (producer before consumer) if you are using local variables, which then become channels.
   c. Use read before write (consumer before producer) if you are using function arguments. Any intra-body anti-dependencies must be preserved by the design.
   d. Function return type must be void.
   e. No loop-carried dependencies among different processes via variables.
      - Inside the canonical loop (i.e., values written by one iteration and read by a following one).
      - Among successive calls to the top function (i.e., inout argument written by one iteration and read by the following iteration).

Dataflow Checking

Vitis HLS has a dataflow checker which, when enabled, checks the code to see if it is in the recommended canonical form. Otherwise it will emit an error/warning message to the user. By default this checker is set to `warning` . You can set the checker to `error` or disable it by selecting `off` in the strict mode of the `config_dataflow` TCL command:

```
config_dataflow -strict_mode  (off | error | warning)
```

Dataflow Optimization Limitations

The `DATAFLOW` optimization optimizes the flow of data between tasks (functions and loops), and ideally pipelined functions and loops for maximum performance. It does not require these tasks to be chained, one after the other, however there are some limitations in how the data is transferred.

The following behaviors can prevent or limit the overlapping that Vitis HLS can perform with `DATAFLOW` optimization:

- Single-producer-consumer violations
- Bypassing tasks
- Feedback between tasks
- Conditional execution of tasks
- Loops with multiple exit conditions

IMPORTANT: If any of these coding styles are present, Vitis HLS issues a message describing the situation.

Note: You can use the dataflow viewer in the Analysis perspective to view the structure when the `DATAFLOW` directive is applied.

Single-producer-consumer Violations

For Vitis HLS to perform the `DATAFLOW` optimization, all elements passed between tasks must follow a single-producer-consumer model. Each variable must be driven from a single task and only be consumed by a single task. In the following code example, `temp1` fans out and is consumed by both `Loop2` and `Loop3`. This violates the single-producer-consumer model.

```
void foo(int data_in[N< N; i++) {
 temp1[i] = data_in[i] * scale;
 temp2[i] = data_in[i] >> scale;
 }
 Loop2: for(int j = 0; j < N; j++) {
 temp3[j] = temp1[j] + 123;
 }
 Loop3: for(int k = 0; k < N; k++) {
 data_out[k] = temp2[k] + temp3[k];
 }
}
```

A modified version of this code uses function `Split` to create a single-producer-consumer design. In this case, data flows from void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) { int temp1[N]; Loop1: for(int i = 0; i < N; i++) { temp1[i] = data_in[i] * scale; } Loop2: for(int j = 0; j < N; j++) { data_out1[j] = temp1[j] * 123; } Loop3: for(int k = 0; k < N; k++) { data_out2[k] = temp1[k] * 456; } } `Loop1` to function `Split` and then to `Loop2` and `Loop3`. The data now flows between all four tasks, and Vitis HLS can perform the `DATAFLOW` optimization.

```
void Split (in[N], out1[N], out2[N]) {
// Duplicated data
 L1:for(int i=1;i<N;i++) {
 out1[i] = in[i];
 out2[i] = in[i];
 }
}
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

 int temp1[N], temp2[N]. temp3[N];
 Loop1: for(int i = 0; i < N; i++) {
 temp1[i] = data_in[i] * scale;
 }
 Split(temp1, temp2, temp3);
 Loop2: for(int j = 0; j < N; j++) {
 data_out1[j] = temp2[j] * 123;
 }
 Loop3: for(int k = 0; k < N; k++) {
 data_out2[k] = temp3[k] * 456;
 }
}
```

Bypassing Tasks

In addition, data should generally flow from one task to another. If you bypass tasks, this can reduce the performance of the `DATAFLOW` optimization. In the following example, `Loop1` generates the values for `temp1` and `temp2`. However, the next task, `Loop2`, only uses the value of `temp1`. The value of `temp2` is not consumed until after `Loop2`. Therefore, `temp2` bypasses the next task in the sequence, which can limit the performance of the `DATAFLOW` optimization.

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {

 int temp1[N], temp2[N
```

In this case, you should increase the depth of the PIPO buffer used to store `temp2` to be 3, instead of the default depth of 2. This lets the buffer store the value intended for `Loop3`, while `Loop2` is being executed. Similarly, a PIPO that bypasses two processes should have a depth of 4. Set the depth of the buffer with the STREAM pragma or directive:

```
#pragma HLS STREAM off variable=temp2 depth=3
```

Feedback Between Tasks

Feedback occurs when the output from a task is consumed by a previous task in the DATAFLOW region. Feedback between tasks is not permitted in a `DATAFLOW` region. When Vitis HLS detects feedback, it issues a warning, depending on the situation, and might not perform the `DATAFLOW` optimization.

Conditional Execution of Tasks

The `DATAFLOW` optimization does not optimize tasks that are conditionally executed. The following example highlights this limitation. In this example, the conditional execution of `Loop1` and `Loop2` prevents Vitis HLS from optimizing the data flow between these loops, because the data does not flow from one loop into the next.

```
void foo(int data_in1[N], int data_out[N], int sel) {

 int temp1[N], temp2[N];

 if (sel) {
 Loop1: for(int i = 0; i < N; i++) {
 temp1[i] = data_in[i] * 123;
 temp2[i] = data_in[i];
 }
 } else {
 Loop2: for(int j = 0; j < N; j++) {
 temp1[j] = data_in[j] * 321;
 temp2[j] = data_in[j];
 }
 }
 Loop3: for(int k = 0; k < N; k++) {
 data_out[k] = temp1[k] * temp2[k];
 }
}
```

To ensure each loop is executed in all cases, you must transform the code as shown in the following example. In this example, the conditional statement is moved into the first loop. Both loops are always executed, and data always flows from one loop to the next.

```
void foo(int data_in[N], int data_out[N], int sel) {

 int temp1[N], temp2[N];

 Loop1: for(int i = 0; i < N; i++) {
 if (sel) {
 temp1[i] = data_in[i] * 123;
 } else {
 temp1[i] = data_in[i] * 321;
 }
 }
 Loop2: for(int j = 0; j < N; j++) {
 temp2[j] = data_in[j];
 }
 Loop3: for(int k = 0; k < N; k++) {
 data_out[k] = temp1[k] * temp2[k];
 }
}
```

Loops with Multiple Exit Conditions

Loops with multiple exit points cannot be used in a `DATAFLOW` region. In the following example, `Loop2` has three exit conditions:

- An exit defined by the value of `N`; the loop will exit when `k>=N`.
- An exit defined by the `break` statement.
- An exit defined by the `continue` statement.

```
#include "ap_cint.h"
#define N 16

typedef int8 din_t;
typedef int15 dout_t;
typedef uint8 dsc_t;
typedef uint1 dsel_t;

void multi_exit(din_t data_in[N], dsc_t scale, dsel_t select, dout_t data_out[N]) {
 dout_t temp1[N], temp2[N];
 int i,k;

 Loop1: for(i = 0; i < N; i++) {
 temp1[i] = data_in[i] * scale;
 temp2[i] = data_in[i] >> scale;
 }

 Loop2: for(k = 0; k < N; k++) {
 switch(select) {
        case  0: data_out[k] = temp1[k] + temp2[k];
        case  1: continue;
        default: break;
 }
 }
}
```

Because a loop's exit condition is always defined by the loop bounds, the use of `break` or `continue` statements will prohibit the loop being used in a `DATAFLOW` region.

Finally, the `DATAFLOW` optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the `DATAFLOW` optimization, you must apply the `DATAFLOW` optimization to the loop, the sub-function, or inline the sub-function.

You can also use `std::complex` inside the `DATAFLOW` region. However, they should be used with an `__attribute__((no_ctor))` as shown in the following example:

```
void proc_1(std::complex<float> (&buffer)[50], const std::complex<float> *in);
void proc_2(hls::Stream<std::complex<float>> &fifo, const std::complex<float> (&buffer)[50],
std::complex<float> &acc);
void proc_3(std::complex<float> *out, hls::Stream<std::complex<float>> &fifo, const std::complex<float> acc);

void top(std::complex<float> *out, const std::complex<float> *in) {
#pragma HLS DATAFLOW

  std::complex<float> acc __attribute((no_ctor)); // Here
  std::complex<float> buffer[50] __attribute__((no_ctor)); // Here
  hls::Stream<std::complex<float>, 5> fifo; // Not here

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

Configuring Dataflow Memory Channels

Vitis HLS implements channels between the tasks as either ping-pong or FIFO buffers, depending on the access patterns of the producer and the consumer of the data:

- For scalar, pointer, and reference parameters, Vitis HLS implements the channel as a FIFO.
- If the parameter (producer or consumer) is an array, Vitis HLS implements the channel as a ping-pong buffer or a FIFO as follows:
    - If Vitis HLS determines the data is accessed in sequential order, Vitis HLS implements the memory channel as a FIFO channel of depth 2.
    - If Vitis HLS is unable to determine that the data is accessed in sequential order or determines the data is accessed in an arbitrary manner, Vitis HLS implements the memory channel as a ping-pong buffer, that is, as two block RAMs each defined by the maximum size of the consumer or producer array.

      Note: A ping-pong buffer ensures that the channel always has the capacity to hold all samples without a loss. However, this might be an overly conservative approach in some cases.

To explicitly specify the default channel used between tasks, use the `config_dataflow` configuration. This configuration sets the default channel for all channels in a design. To reduce the size of the memory used in the channel and allow for overlapping within an iteration, you can use a FIFO. To explicitly set the depth (i.e., number of elements) in the FIFO, use the `-fifo_depth` option.

Specifying the size of the FIFO channels overrides the default approach. If any task in the design can produce or consume samples at a greater rate than the specified size of the FIFO, the FIFOs might become empty (or full). In this case, the design halts operation, because it is unable to read (or write). This might result in or lead to a stalled, deadlock state.

Note: If a deadlocked situation is created, you will only see this when executing C/RTL co-simulation or when the block is used in a complete system.

When setting the depth of the FIFOs, Xilinx recommends initially setting the depth as the maximum number data values transferred (e.g., the size of the array passed between tasks), confirming the design passes C/RTL co-simulation, and then reducing the size of the FIFOs and confirming C/RTL co-simulation still completes without issues. If RTL co-simulation fails, the size of the FIFO is likely too small to prevent stalling or a deadlock situation.

The Vitis HLS GUI can display a histogram of the occupation of each FIFO/PIPO buffer over time, after RTL co-simulation has been run. This can be useful to help determine the best depth for each buffer.

Specifying Arrays as Ping-Pong Buffers or FIFOs

All arrays are implemented by default as ping-pong to enable random access. These buffers can also be sized if needed. For example, in some circumstances, such as when a task is being bypassed, a performance degradation is possible. To mitigate this affect on performance, you can give more slack to the producer and consumer by increasing the size of these buffers by using the STREAM directive as shown below.

```
void top ( ... ) {
#pragma HLS dataflow
  int A[1024];
#pragma HLS stream off variable=A depth=3

  producer(A, B, …);  // producer writes A and B
  middle(B, C, ...);  // middle reads B and writes C
  consumer(A, C, …);  // consumer reads A and C
```

In the interface, arrays are automatically specified as streaming if an array on the top-level function interface is set as interface type `ap_fifo`, `axis` or `ap_hs`, it is automatically set as streaming.

Inside the design, all arrays must be specified as streaming using the STREAM directive if a FIFO is desired for the implementation.

Note: When the STREAM directive is applied to an array, the resulting FIFO implemented in the hardware contains as many elements as the array. The `-depth` option can be used to specify the size of the FIFO.

The STREAM directive is also used to change any arrays in a DATAFLOW region from the default implementation specified by the `config_dataflow` configuration.

- If the `config_dataflow default_channel` is set as ping-pong, any array can be implemented as a FIFO by applying the STREAM directive to the array.

  Note: To use a FIFO implementation, the array must be accessed in a streaming manner.

- If the `config_dataflow default_channel` is set to FIFO or Vitis HLS has automatically determined the data in a DATAFLOW region is accessed in a streaming manner, any array can still be implemented as a ping-pong implementation by applying the STREAM directive to the array with the `-off` option.

IMPORTANT: To preserve the accesses, it might be necessary to prevent compiler optimizations (dead code elimination particularly) by using the volatile qualifier.

When an array in a DATAFLOW region is specified as streaming and implemented as a FIFO, the FIFO is typically not required to hold the same number of elements as the original array. The tasks in a DATAFLOW region consume each data sample as soon as it becomes available. The `config_dataflow` command with the `-fifo_depth` option or the STREAM directive with the `-depth` can be used to set the size of the FIFO to the minimum number of elements required to ensure flow of data never stalls. If the `-off` option is selected, the `-off` option sets the depth (number of blocks) of the ping-pong. The depth should be at least 2.

Specifying Compiler-FIFO Depth

Start Propagation

The compiler might automatically create a **start FIFO** to propagate a **start token** to an internal process. Such FIFOs can sometimes be a bottleneck for performance, in which case you can increase the default size (fixed to 2) with the following command:

```
config_dataflow -start_fifo_depth <value>
```

If an unbounded slack between producer and consumer is needed, and internal processes can run forever, fully and safely driven by their inputs or outputs (FIFOs or PIPOs), these start FIFOs can be removed, at user's risk, locally for a given dataflow region with the pragma:

```
#pragma HLS DATAFLOW disable_start_propagation
```

Scalar Propagation

The compiler automatically propagates some scalars from C/C++ code through **scalar FIFOs** between processes. Such FIFOs can sometimes be a bottleneck for performance or cause deadlocks, in which case you can set the size (the default value is set to `-fifo_depth`) with the following command:

```
config_dataflow -scalar_fifo_depth <value>
```

Stable Arrays

The `stable` pragma can be used to mark input or output variables of a dataflow region. Its effect is to remove their corresponding synchronizations, assuming that the user guarantees this removal is indeed correct.

```
void dataflow_region(int A[...], ...
#pragma HLS stable variable=A
#pragma HLS dataflow
    proc1(...);
    proc2(A, ...);
```

Without the `stable` pragma, and assuming that `A` is read by `proc2`, then `proc2` would be part of the initial synchronization (via `ap_start`), for the dataflow region where it is located. This means that `proc1` would not restart until `proc2` is also ready to start again, which would prevent dataflow iterations to be overlapped and induce a possible loss of performance. The `stable` pragma indicates that this synchronization is not necessary to preserve correctness.

In the previous example, without the `stable` pragma, and assuming that `A` is read by `proc2` as `proc2` is bypassing the tasks, there will be a performance loss.

With the `stable` pragma, the compiler assumes that:

- if `A` is read by `proc2`, then the memory locations that are read will not be overwritten, by any other process or calling context, while `dataflow_region` is being executed.
- if `A` is written by `proc2`, then the memory locations written will not be read, before their definition, by any other process or calling context, while `dataflow_region` is being executed.

A typical scenario is when the caller updates or reads these variables only when the dataflow region has not started yet or has completed execution.

Using ap_ctrl_none Inside the Dataflow

The `ap_ctrl_none` block-level I/O protocol avoids the rigid synchronization scheme implied by the `ap_ctrl_hs` and `ap_ctrl_chain` protocols. These protocols require that all processes in the region are executed exactly the same number of times in order to better match the C behavior.

However, there are situations where, for example, the intent is to have a faster process that executes more frequently to distribute work to several slower ones.

For any dataflow region (except "dataflow-in-loop"), it is possible to specify

```
#pragma HLS interface ap_ctrl_none port=return
```

as long as all the following conditions are satisfied:

- The region and all the processes it contains communicates only via FIFOs (hls::stream, streamed arrays, AXIS); that is, excluding memories.
- All the parents of the region, up to the top level design, must fit the following requirements:
  - They must be dataflow regions (excluding "dataflow-in-loop").
  - They must all specify `ap_ctrl_none`.

This means that none of the parents of a dataflow region with `ap_ctrl_none` in the hierarchy can be:

- A sequential or pipelined FSM

- A dataflow region inside a for loop ("dataflow-in-loop")

The result of this pragma is that `ap_ctrl_chain` is not used to synchronize any of the processes inside that region. They are executed or stalled based on the availability of data in their input FIFOs and space in their output FIFOs. For example:

```
void region(...) {
#pragma HLS dataflow
#pragma HLS interface ap_ctrl_none port=return
    hls::stream<int> outStream1, outStream2;
    demux(inStream, outStream1, outStream2);
    worker1(outStream1, ...);
    worker2(outStream2, ....);
```

In this example, `demux` can be executed twice as frequently as `worker1` and `worker2`. For example, it can have II=1 while `worker1` and `worker2` can have II=2, and still achieving a global II=1 behavior.

Note:

- Non-blocking reads may need to be used very carefully inside processes that are executed less frequently to ensure that C simulation works.
- The pragma is applied to a region, not to the individual processes inside it.
- Deadlock detection must be disabled in co-simulation. This can be done with the `-disable_deadlock_detection` option in cosim_design (dfh1584824522428.html).

Optimizing for Latency

Using Latency Constraints

Vitis HLS supports the use of a latency constraint on any scope. Latency constraints are specified using the LATENCY directive.

When a maximum and/or minimum LATENCY constraint is placed on a scope, Vitis HLS tries to ensure all operations in the function complete within the range of clock cycles specified.

The latency directive applied to a loop specifies the required latency for a single iteration of the loop: it specifies the latency for the loop body, as the following examples shows:

```
Loop_A: for (i=0; i<N; i++) {
#pragma HLS latency max=10
  ..Loop Body...
}
```

If the intention is to limit the total latency of all loop iterations, the latency directive should be applied to a region that encompasses the entire loop, as in this example:

```
Region_All_Loop_A: {
#pragma HLS latency max=10
Loop_A: for (i=0; i<N; i++)
  {
  ..Loop Body...
  }
}
```

In this case, even if the loop is unrolled, the latency directive sets a maximum limit on all loop operations.

If Vitis HLS cannot meet a maximum latency constraint it relaxes the latency constraint and tries to achieve the best possible result.

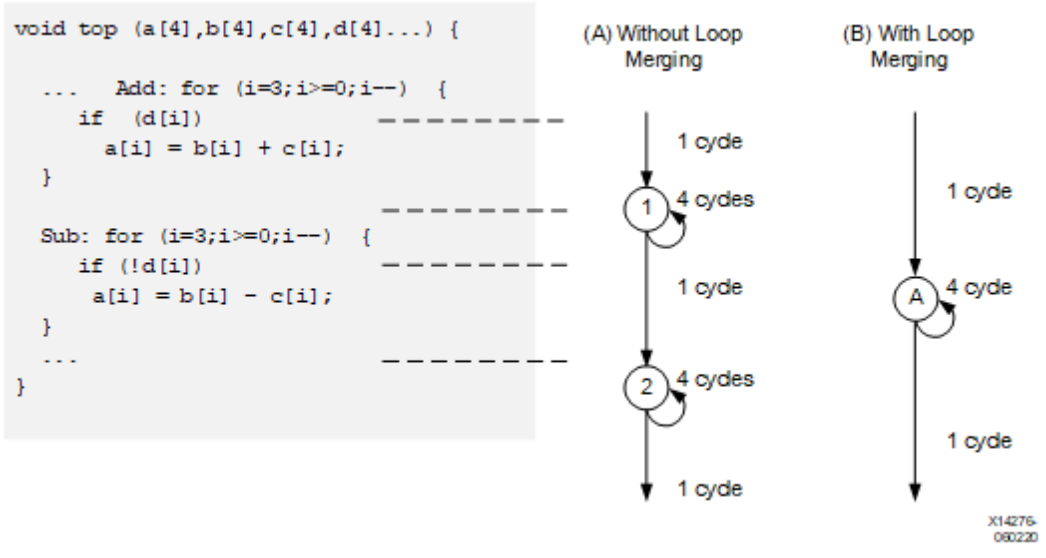
If a minimum latency constraint is set and Vitis HLS can produce a design with a lower latency than the minimum required it inserts dummy clock cycles to meet the minimum latency.

Merging Sequential Loops to Reduce Latency

All rolled loops imply and create at least one state in the design FSM. When there are multiple sequential loops it can create additional unnecessary clock cycles and prevent further optimizations.

The following figure shows a simple example where a seemingly intuitive coding style has a negative impact on the performance of the RTL design.

Figure 41: Loop Directives



In the preceding figure, (A) shows how, by default, each rolled loop in the design creates at least one state in the FSM. Moving between those states costs clock cycles: assuming each loop iteration requires one clock cycle, it take a total of 11 cycles to execute both loops:

- 1 clock cycle to enter the ADD loop.
- 4 clock cycles to execute the add loop.
- 1 clock cycle to exit ADD and enter SUB.
- 4 clock cycles to execute the SUB loop.
- 1 clock cycle to exit the SUB loop.
- For a total of 11 clock cycles.

In this simple example it is obvious that an else branch in the ADD loop would also solve the issue but in a more complex example it may be less obvious and the more intuitive coding style may have greater advantages.

The LOOP_MERGE optimization directive is used to automatically merge loops. The LOOP_MERGE directive will seek so to merge all loops within the scope it is placed. In the above example, merging the loops creates a control structure similar to that shown in (B) in the preceding figure, which requires only 6 clocks to complete.

Merging loops allows the logic within the loops to be optimized together. In the example above, using a dual-port block RAM allows the add and subtraction operations to be performed in parallel.

Currently, loop merging in Vitis HLS has the following restrictions:

- If loop bounds are all variables, they must have the same value.
- If loops bounds are constants, the maximum constant value is used as the bound of the merged loop.
- Loops with both variable bound and constant bound cannot be merged.
- The code between loops to be merged cannot have side effects: multiple execution of this code should generate the same results (a=b is allowed, a=a+1 is not).
- Loops cannot be merged when they contain FIFO accesses: merging would change the order of the reads and writes from a FIFO: these must always occur in sequence.

Flattening Nested Loops to Improve Latency

In a similar manner to the consecutive loops discussed in the previous section, it requires additional clock cycles to move between rolled nested loops. It requires one clock cycle to move from an outer loop to an inner loop and from an inner loop to an outer loop.

In the small example shown here, this implies 200 extra clock cycles to execute loop `Outer`.

```
void foo_top { a, b, c, d} {
 ...
 Outer: while(j<100)
 Inner: while(i<6) // 1 cycle to enter inner
 ...
 LOOP_BODY
 ...
 } // 1 cycle to exit inner
 }
 ...
}
```

Vitis HLS provides the `set_directive_loop_flatten` command to allow labeled perfect and semi-perfect nested loops to be flattened, removing the need to re-code for optimal hardware performance and reducing the number of cycles it takes to perform the operations in the loop.

Perfect loop nest

Only the innermost loop has loop body content, there is no logic specified between the loop statements and all the loop bounds are constant.

Semi-perfect loop nest:

Only the innermost loop has loop body content, there is no logic specified between the loop statements but the outermost loop bound can be a variable.

For imperfect loop nests, where the inner loop has variables bounds or the loop body is not exclusively inside the inner loop, designers should try to restructure the code, or unroll the loops in the loop body to create a perfect loop nest.

When the directive is applied to a set of nested loops it should be applied to the inner most loop that contains the loop body.

```
set_directive_loop_flatten top/Inner
```

Loop flattening can also be performed using the directive tab in the GUI, either by applying it to individual loops or applying it to all loops in a function by applying the directive at the function level.

Optimizing for Area

Data Types and Bit-Widths

The bit-widths of the variables in the C function directly impact the size of the storage elements and operators used in the RTL implementation. If a variables only requires 12-bits but is specified as an integer type (32-bit) it will result in larger and slower 32-bit operators being used, reducing the number of operations that can be performed in a clock cycle and potentially increasing initiation interval and latency.

- Use the appropriate precision for the data types.
- Confirm the size of any arrays that are to be implemented as RAMs or registers. The area impact of any over-sized elements is wasteful in hardware resources.
- Pay special attention to multiplications, divisions, modulus or other complex arithmetic operations. If these variables are larger than they need to be, they negatively impact both area and performance.

Function Inlining

Function inlining removes the function hierarchy. A function is inlined using the INLINE directive. Inlining a function may improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function. This type of function inlining is also performed automatically by Vitis HLS. Small functions are automatically inlined.

Inlining allows functions sharing to be better controlled. For functions to be shared they must be used within the same level of hierarchy. In this code example, function `foo_top` calls `foo` twice and function `foo_sub` .

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

Inlining function `foo_sub` and using the ALLOCATION directive to specify only 1 instance of function `foo` is used, results in a design which only has one instance of function foo: one-third the area of the example above.

```
foo_sub (p, q) {
#pragma HLS INLINE
 int q1 = q + 10;
 foo(p1,q); // foo_3
 ...
}
void foo_top { a, b, c, d} {
#pragma HLS ALLOCATION instances=foo limit=1 function
 ...
 foo(a,b); //foo_1
 foo(a,c); //foo_2
 foo_sub(a,d);
 ...
}
```

The INLINE directive optionally allows all functions below the specified function to be recursively inlined by using the `recursive` option. If the `recursive` option is used on the top-level function, all function hierarchy in the design is removed.

The INLINE `off` option can optionally be applied to functions to prevent them being inlined. This option may be used to prevent Vitis HLS from automatically inlining a function.

The INLINE directive is a powerful way to substantially modify the structure of the code without actually performing any modifications to the source code and provides a very powerful method for architectural exploration.

Array Reshaping

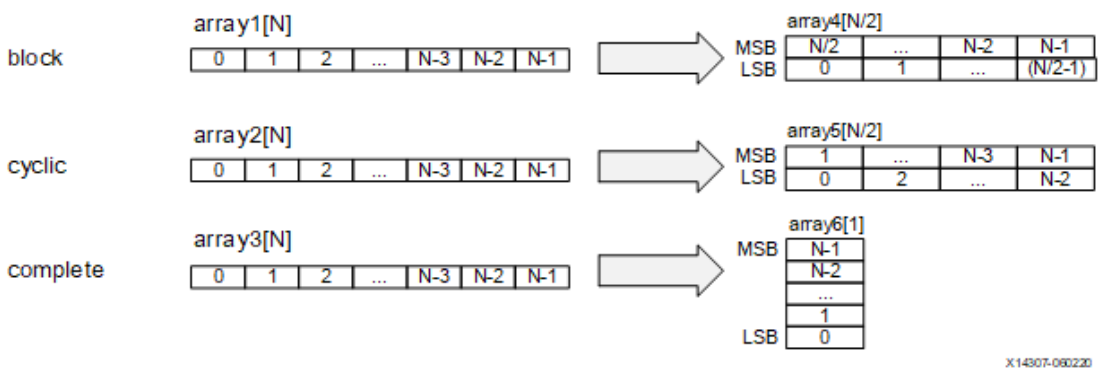
The ARRAY_RESHAPE directive combines ARRAY_PARTITIONING with a vertical mode of remapping, and is used to reduce the number of block RAM while still allowing the beneficial attributes of partitioning: parallel access to the data.

Given the following example code:

```
void foo (...) {
int  array1[N];
int  array2[N];
int  array3[N];
#pragma HLS ARRAY_RESHAPE variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array2 cycle factor=2 dim=1
#pragma HLS ARRAY_RESHAPE variable=array3 complete dim=1
...
}
```

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

Figure 42: Array Reshaping



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

```
config_unroll -tripcount_threshold 16
```

Controlling Hardware Resources

During synthesis, Vitis HLS performs the following basic tasks:

- Elaborates the C, C++ source code into an internal database containing the operators in the C code, such as additions, multiplications, array reads, and writes.
- Maps the operators onto implementations in the hardware.

  Implementations are the specific hardware components used to create the design (such as adders, multipliers, pipelined multipliers, and block RAM).

Commands, pragmas and directives provide control over each of these steps, allowing you to control the hardware implementation at a fine level of granularity.

Limiting the Number of Operators

Explicitly limiting the number of operators to reduce area may be required in some cases: the default operation of Vitis HLS is to first maximize performance. Limiting the number of operators in a design is a useful technique to reduce the area of the design: it helps reduce area by forcing the sharing of operations. However, this might cause a decline in performance.

The `ALLOCATION` directive allows you to limit how many operators are used in a design. For example, if a design called `foo` has 317 multiplications but the FPGA only has 256 multiplier resources (DSP48s). The `ALLOCATION` pragma shown below directs Vitis HLS to create a design with a maximum of 256 multiplication (`mul`) operators:

```
dout_t array_arith (dio_t d[317]) {
 static int acc;
 int i;
#pragma HLS ALLOCATION instances=mul limit=256 operation

 for (i=0;i<317;i++) {
#pragma HLS UNROLL
 acc += acc * d[i];
 }
 rerun acc;
}
```

Note: If you specify an `ALLOCATION` limit that is greater than needed, Vitis HLS attempts to use the number of resources specified by the limit, or the maximum necessary, which reduces the amount of sharing.

You can use the `type` option to specify if the ALLOCATION directives limits operations, implementations, or functions. The following table lists all the operations that can be controlled using the ALLOCATION directive.

Note: The operations listed below are supported by the ALLOCATION pragma or directive. The BIND_OP pragma or directive supports a subset of operators as described in the command syntax.

Table 9. Vitis HLS Operators

Operator	Description
add	Integer Addition
ashr	Arithmetic Shift-Right
dadd	Double-precision floating point addition
dcmp	Double -precision floating point comparison
ddiv	Double -precision floating point division
dmul	Double -precision floating point multiplication
drecip	Double -precision floating point reciprocal
drem	Double -precision floating point remainder
drsqrt	Double -precision floating point reciprocal square root
dsub	Double -precision floating point subtraction
dsqrt	Double -precision floating point square root
fadd	Single-precision floating point addition
fcmp	Single-precision floating point comparison
fdiv	Single-precision floating point division
fmul	Single-precision floating point multiplication
frecip	Single-precision floating point reciprocal
frem	Single-precision floating point remainder
frsqrt	Single-precision floating point reciprocal square root
fsub	Single-precision floating point subtraction
fsqrt	Single-precision floating point square root
icmp	Integer Compare
lshr	Logical Shift-Right
mul	Multiplication
sdiv	Signed Divider
shl	Shift-Left

Operator	Description
srem	Signed Remainder
sub	Subtraction
udiv	Unsigned Division
urem	Unsigned Remainder

Controlling Hardware Implementation

When synthesis is performed, Vitis HLS uses the timing constraints specified by the clock, the delays specified by the target device together with any directives specified by you, to determine which hardware implementations to use for various operators in the code. For example, to implement a multiplier operation, Vitis HLS could use the combinational multiplier or use a pipeline multiplier.

The implementations which are mapped to operators during synthesis can be limited by specifying the ALLOCATION pragma or directive, in the same manner as the operators. Instead of limiting the total number of multiplication operations, you can choose to limit the number of combinational multipliers, forcing any remaining multiplications to be performed using pipelined multipliers (or vice versa).

The BIND_OP or BIND_STORAGE pragmas or directives are used to explicitly specify which implementations to use for specific operations or storage types. The following command informs Vitis HLS to use a 2-stage pipelined multiplier for variable `c`. It is left to Vitis HLS which implementation to use for variable `d`.

```
int foo (int a, int b) {
 int c, d;
#pragma HLS BIND_OP variable=c op=mul latency=2
 c = a*b;
 d = a*c;

 return d;
}
```

In the following example, the BIND_OP pragma specifies that the add operation for variable `temp` is implemented using the `dsp` implementation. This ensures that the operation is implemented using a DSP module primitive in the final design. By default, add operations are implemented using LUTs.

```
void apint_arith(dinA_t  inA, dinB_t  inB,
          dout1_t *out1
  ) {

 dout2_t temp;
#pragma HLS BIND_OP variable=temp op=add impl=dsp

 temp = inB + inA;
 *out1 = temp;

}
```

Refer to the BIND_OP or BIND_STORAGE pragmas or directives to obtain details on the implementations available for assignment to operations or storage types.

In the following example, the BIND_OP pragma specifies the multiplication for `out1` is implemented with a 3-stage pipelined multiplier.

```
void foo(...) {
#pragma HLS BIND_OP variable=out1 op=mul latency=3

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

}
```

If the assignment specifies multiple identical operators, the code must be modified to ensure there is a single variable for each operator to be controlled. For example, in the following code, if only the first multiplication (`inA * inB`) is to be implemented with a pipelined multiplier:

```
*out1 = inA * inB * inC;
```

The code should be changed to the following with the pragma specified on the `Result_tmp` variable:

```
#pragma HLS BIND_OP variable=Result_tmp op=mul latency=3
 Result_tmp = inA * inB;
 *out1 = Result_tmp * inC;
```

Optimizing Logic

Inferring Shift Registers

Vitis HLS will now infer a shift register when encountering the following code:

```
int A[N]; // This will be replaced by a shift register

for(...) {
  // The loop below is the shift operation
  for (int i = 0; i < N-1; ++i)
    A[i] = A[i+1];
  A[N] = ...;

  // This is an access to the shift register
  ... A[x] ...
}
```

Shift registers can perform a shift/cycle, which offers a significant performance improvement, and also allows a random read access per cycle anywhere in the shift register, thus it is more flexible than a FIFO.

Controlling Operator Pipelining

Vitis HLS automatically determines the level of pipelining to use for internal operations. You can use the BIND_OP or BIND_STORAGE pragmas with the `-latency` option to explicitly specify the number of pipeline stages and override the number determined by Vitis HLS.

RTL synthesis might use the additional pipeline registers to help improve timing issues that might result after place and route. Registers added to the output of the operation typically help improve timing in the output datapath. Registers added to the input of the operation typically help improve timing in both the input datapath and the control logic from the FSM.

You can use the `config_op` command to pipeline all instances of a specific operation used in the design that have the same pipeline depth. Refer to config_op (arv1584806222219.html) for more information.

Optimizing Logic Expressions

During synthesis several optimizations, such as strength reduction and bit-width minimization are performed. Included in the list of automatic optimizations is expression balancing.

Expression balancing rearranges operators to construct a balanced tree and reduce latency.

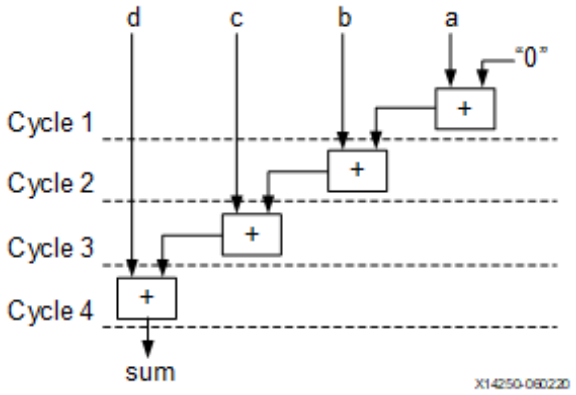
- For integer operations expression balancing is on by default but may be disabled.
- For floating-point operations, expression balancing is off by default but may be enabled.

Given the highly sequential code using assignment operators such as `+=` and `*=` in the following example:

```
data_t foo_top (data_t a, data_t b, data_t c, data_t d)
{
 data_t sum;

 sum = 0;
 sum += a;
 sum += b;
 sum += c;
 sum += d;
 return sum;

}
```

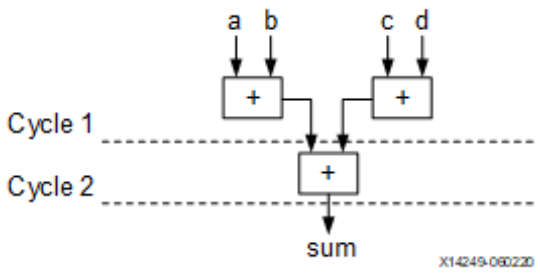
Without expression balancing, and assuming each addition requires one clock cycle, the complete computation for `sum` requires four clock cycles shown in the following figure.

Figure 43: Adder Tree



However additions `a+b` and `c+d` can be executed in parallel allowing the latency to be reduced. After balancing the computation completes in two clock cycles as shown in the following figure. Expression balancing prohibits sharing and results in increased area.

Figure 44: Adder Tree After Balancing



For integers, you can disable expression balancing using the EXPRESSION_BALANCE optimization directive with the `off` option. By default, Vitis HLS does not perform the EXPRESSION_BALANCE optimization for operations of type `float` or `double`. When synthesizing `float` and `double` types, Vitis HLS maintains the order of operations performed in the C code to ensure that the results are the same as the C simulation. For example, in the following code example, all variables are of type `float` or `double`. The values of `O1` and `O2` are not the same even though they appear to perform the same basic calculation.

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

This behavior is a function of the saturation and rounding in the C standard when performing operation with types `float` or `double`. Therefore, Vitis HLS always maintains the exact order of operations when variables of type `float` or `double` are present and does not perform expression balancing by default.

You can enable expression balancing with `float` and `double` types using the configuration `config_compile` option as follows:

1. Select **Solution** > (and then)**Solution Settings**.
2. In the Solution Settings dialog box, click the **General** category, and click **Add**.
3. In the Add Command dialog box, select **config_compile**, and enable **unsafe_math_optimizations**.

With this setting enabled, Vitis HLS might change the order of operations to produce a more optimal design. However, the results of C/RTL cosimulation might differ from the C simulation.

The `unsafe_math_optimizations` feature also enables the `no_signed_zeros` optimization. The `no_signed_zeros` optimization ensures that the following expressions used with float and double types are identical:

```
x - 0.0 = x;
x + 0.0 = x;
0.0 - x = -x;
x - x = 0.0;
x*0.0 = 0.0;
```

Without the `no_signed_zeros` optimization the expressions above would not be equivalent due to rounding. The optimization may be optionally used without expression balancing by selecting only this option in the `config_compile` configuration.

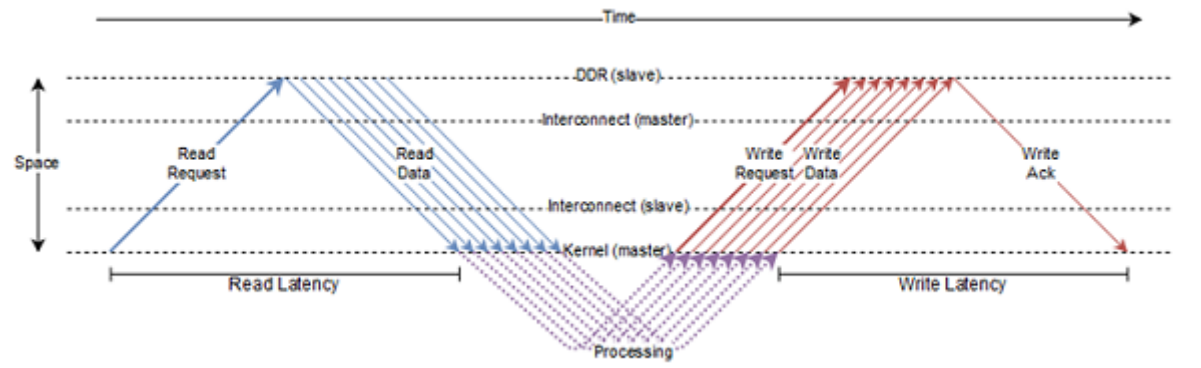
TIP: When the `unsafe_math_optimizations` and `no_signed_zero` optimizations are used, the RTL implementation will have different results than the C simulation. The test bench should be capable of ignoring minor differences in the result: check for a range, do not perform an exact comparison.

Optimizing Burst Transfers

Overview of Burst Transfers

In terms of optimization techniques, you should perform burst optimizations only after you have exhausted all the normal options for optimizing your HLS kernel. Bursting is an optimization that tries to intelligently aggregate your memory accesses to the DDR to maximize the throughput bandwidth and/or minimize the latency. Bursting is one of the many optimizations to the kernel. Bursting typically gives you a 4-5X improvement while other optimizations, like access widening or ensuring there are no dependencies through the DDR, can provide even bigger performance improvements. Typically, bursting is useful when you have contention on the DDR ports from multiple competing kernels.

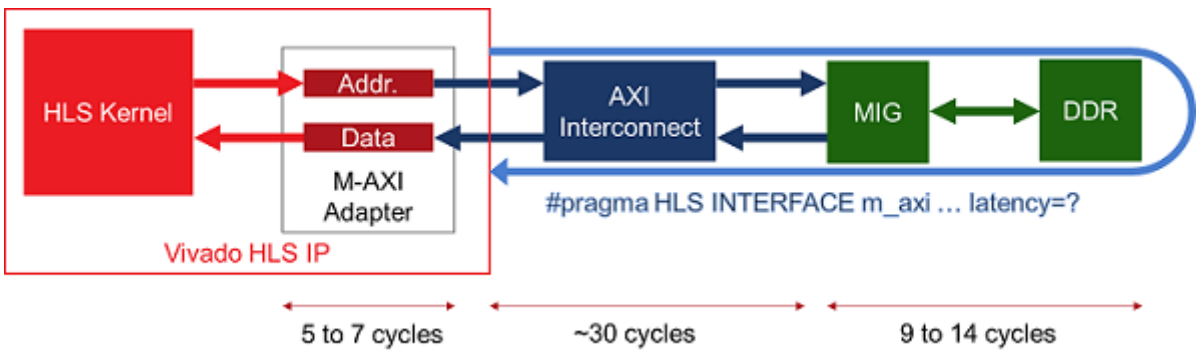
Figure 45: AXI Protocol



The figure above shows how the AXI protocol works. The HLS kernel sends out a read request for a burst of length 8 and then sends a write request burst of length 8. The read latency is defined as the time taken between the sending of the read request burst to when the data from the first read request in the burst is received by the kernel. Similarly, the write latency is defined as the time taken between when data for the last write in the write burst is sent and the time the write acknowledgment is received by the kernel. Read requests are usually sent at the first available opportunity while write requests get queued until the data for each write in the burst becomes available.

To help you understand the various latencies that are possible in the system, the following figure shows what happens when an HLS kernel sends a burst to the DDR.

Figure 46: Burst Transaction Diagram



When your design makes a read/write request, the request is sent to the DDR via several specialized helper modules. First is the M-AXI adapter that serves as a buffer for the requests created by the HLS kernel. The adapter contains logic to cut large bursts into smaller ones (which it needs to do to prevent hogging the channel or if the request crosses the 4 KB boundary (https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf)), and can also stall the sending of burst requests (depending on the maximum outstanding requests parameter) so that it can safely buffer the entirety of the data for each kernel. Getting through the adapter will cost a few cycles of latency, typically 5 to 7 cycles. Then the request goes to the AXI interconnect that routes the kernel's request to the MIG and then eventually to the DDR. Getting through the interconnect is expensive in latency and can take around 30 cycles. Finally, getting to the DDR and back can cost anywhere from 9 to 14 cycles. These are not precise measurements of latency but rather estimates provided to show the relative latency cost of these specialized modules. For more precise measurements, you can test and observe these latencies using the Application Timeline report for your specific system.

TIP: For information about the Application Timeline report, see Application Timeline (https://www.xilinx.com/cgi-bin/docs/rdoc?t=vitis+doc;v=2020.1;d=profilingapplication.html;a=prd1522349552911) in the *Vitis Unified Software Platform Documentation* (UG1416).

Another way to view the latencies in the system is as follows: the interconnect has an average II of 2 while the DDR controller has an average II of 4-5 cycles on requests (while on the data they are both II=1). The interconnect arbitration strategy is based on the size of read/write requests, and so data requested with longer burst lengths get prioritized over requests with shorter bursts (thus leading to a bigger channel bandwidth being allocated to longer bursts in case of contention). Of course, a large burst request has the side-effect of preventing anyone else from accessing the DDR, and therefore there must be a compromise between burst length and reducing DDR port contention. Fortunately, the large latencies help prevent some of this port contention, and effective pipelining of the requests can significantly improve the bandwidth throughput available in the system.

Burst Semantics

For a given kernel, the HLS compiler implements the burst analysis optimization as a multi-pass optimization, but on a per function basis. Bursting is only done for a function and bursting across functions is not supported.

At first, the HLS compiler looks for memory accesses in the basic blocks of the function, such as memory accesses in a sequential set of statements inside the function. Assuming the preconditions of bursting are met, each burst inferred in these basic blocks is referred to as **region** burst. The compiler will automatically scan the basic block to build the longest sequence of accesses into a single region burst.

The compiler then looks at loops and tries to infer what are known as **loop** bursts. A loop burst is the sequence of reads/writes across the iterations of a loop. The compiler tries to infer the length of the burst by analyzing the loop induction variable and the trip count of the loop. If the analysis is successful, the compiler can chain the sequences of reads/writes in each iteration of the loop into one long loop burst. The compiler today automatically infers a loop or a region burst, but there is no way to specifically request a loop or a region burst. The code needs to be written so as to cause the tool to infer the loop or region burst.

To understand the underlying semantics of bursting, consider the following code snippet:

```
for(size_t i = 0; i < size; i+=4) {
    out[4*i+0] = f(in[4*i+0]);
    out[4*i+1] = f(in[4*i+1]);
    out[4*i+2] = f(in[4*i+2]);
    out[4*i+3] = f(in[4*i+3]);
}
```

The code above is typically used to perform a series of reads from an array, and a series of writes to an array from within a loop. The code below is what Vitis HLS may infer after performing the burst analysis optimization. Alongside the actual array accesses, the compiler will additionally make the required read and write requests that are necessary for the user selected AXI protocol.

Loop Burst

```
/* requests can move anywhere in func */
rb = ReadReq(in, size);
wb = WriteReq(out, size);
for(size_t i = 0; i < size; i+=4) {
    Write(wb, 4*i+0) = f(Read(rb, 4*i+0));
    Write(wb, 4*i+1) = f(Read(rb, 4*i+1));
    Write(wb, 4*i+2) = f(Read(rb, 4*i+2));
    Write(wb, 4*i+3) = f(Read(rb, 4*i+3));
}
WriteResp(wb);
```

If the compiler can successfully deduce the burst length from the induction variable (`size`) and the trip count of the loop, it will infer one big loop burst and will move the `ReadReq` , `WriteReq` and `WriteResp` calls outside the loop, as shown in the **Loop Burst** example. So, the read requests for all loop iterations are combined into one read request and all the write requests are combined into one write request. Note that all read requests are typically sent out immediately while write requests are only sent out after the data becomes available.

However, if any of the preconditions of bursting are not met, as described in Preconditions and Limitations of Burst Transfer, the compiler may not infer a loop burst but will instead try and infer a region burst where the `ReadReq` , `WriteReg` and `WriteResp` are alongside the read/write accesses being burst optimized, as shown in the **Region Burst** example. In this case, the read and write requests for each loop iteration are combined into one read or write request.

Region Burst

```
for(size_t i = 0; i < size; i+=4) {
    rb = ReadReq(in+4*i, 4);
    wb = WriteReq(out+4*i, 4);
    Write(wb, 0) = f(Read(rb, 0));
    Write(wb, 1) = f(Read(rb, 1));
    Write(wb, 2) = f(Read(rb, 2));
    Write(wb, 3) = f(Read(rb, 3));
    WriteResp(wb);
}
```

Preconditions and Limitations of Burst Transfer

Bursting Preconditions

Bursting is about aggregating successive memory access requests. Here are the set of preconditions that these successive accesses must meet for the bursting optimization to fire successfully:

- Must be all reads, or all writes – bursting reads, and writes is not possible.
- Must be a monotonically increasing order of access (both in terms of the memory location being accessed as well as in time). You cannot access a memory location that is in between two previously accessed memory locations.
- Must be consecutive in memory – one next to another with no gaps or overlap and in forward order.
- The number of read/write accesses (or burst length) must be determinable before the request is sent out. This means that even if the burst length is parametric, it must be computed before the read/write request is sent out.

- If bundling two arrays to the same MAXI port, bursting will be done only for one array, at most, in each direction at any given time.
- There must be no dependency issues from the time a burst request is initiated and finished.

Outer Loop Burst Failure Due to Overlapping Memory Accesses

Outer loop burst inference will fail in the following example because both iteration 0 and iteration 1 of the loop L1 end up accessing the same element in arrays a and b. Important to note that currently, burst inference is an all or nothing type of optimization – i.e. there is no capability for the burst inference to infer a partial burst. It is a greedy algorithm that tries to maximize the length of the burst. An inner loop burst of length 9 will still be inferred in this case.

```
L1: for (int i = 0; i < 8; ++i)
    L2: for (int j = 0; j < 9; ++j)
            b[i*8 + j] = a[i*8 + j];

itr 0: |0 1 2 3 4 5 6 7 8|
itr 1: |               8 9 10 11 12 13 14 15 16|
```

Usage of ap_int/ap_uint Types as Loop Induction Variables

Since the burst inference depends on the loop induction variable and the trip count, using non-native types can hinder the optimization from firing. It is recommended to always use unsigned integer type for the loop induction variable.

Must Enter Loop at Least Once

In some cases, the compiler can fail to infer that the max value of the loop induction variable can never be zero – i.e. if it cannot prove that the loop will always be entered. In such cases, an assert statement will help the compiler infer this.

```
assert (N > 0);
L1: for(int a = 0; a < N; ++a) { … }
```

Inter or Intra Loop Dependencies on Arrays

If you write to an array location and then read from it in the same iteration or the next, this type of array dependency can be hard for the optimization to decipher. Basically, the optimization will fail for these cases because it cannot guarantee that the write will happen before the read.

Conditional Access to Memory

If the memory accesses are being made conditionally, it can cause the burst inferencing algorithm to fail as it cannot reason through the conditional statements. In some cases, the compiler will simplify the conditional and even remove it but it is generally recommended to not use conditional statements around the memory accesses.

M-AXI Accesses Made from Inside a Function Called from a Loop

Cross-functional array access analysis is not a strong suit for compiler transformations such as burst inferencing. In such cases, users can inline the function using the INLINE pragma or directive to avoid burst failures.

```
void my_function(hls::stream<T> &out_pkt, int *din, int input_idx) {
    T v;
    v.data = din[input_idx];
    out_pkt.write(v);
}

void my_kernel(hls::stream<T> &out_pkt,
               int            *din,
               int            num_512_bytes,
               int            num_times) {
#pragma HLS INTERFACE m_axi port = din offset=slave bundle=gmem0
#pragma HLS INTERFACE axis port=out_pkt
#pragma HLS INTERFACE s_axilite port=din bundle=control
#pragma HLS INTERFACE s_axilite port=num_512_bytes bundle=control
#pragma HLS INTERFACE s_axilite port=num_times bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

unsigned int idx = 0;
L0: for (int i = 0; i < ntimes; ++i) {
    L1: for (int j = 0; j < num_512_bytes; ++j) {
#pragma HLS PIPELINE
        my_function(out_pkt, din, idx++);
    }
}
```

Burst inferencing will fail because the memory accesses are being made from a called function. For the burst inferencing to work, it is recommended that users inline any such functions that are making accesses to the M-AXI memory.

An additional reason the burst inferencing will fail in this example is that the memory accessed through `din` in `my_function`, is defined by a variable (`idx`) which is not a function of the loop induction variables `i` and `j`, and therefore may not be sequential or monotonic. Instead of passing `idx`, use `(i*num_512_bytes+j)`.

Loop Burst Inference on a Dataflow Loop

Burst inference is not supported on a loop that has the DATAFLOW pragma or directive. However, each process/task inside the dataflow loop can have bursts. Also, sharing of M-AXI ports is not supported inside a dataflow region because the tasks can execute in parallel.

Options for Controlling AXI4 Burst Behavior

An optimal AXI4 interface is one in which the design never stalls while waiting to access the bus, and after bus access is granted, the bus never stalls while waiting for the design to read/write. To create the optimal AXI4 interface, the following command options are provided in the INTERFACE directive to specify the behavior of the bursts and optimize the efficiency of the AXI4 interface.

Note that some of these options can use internal storage to buffer data and this may have an impact on area and resources:

latency
: Specifies the expected latency of the AXI4 interface, allowing the design to initiate a bus request several cycles (latency) before the read or write is expected. If this figure it too low, the design will be ready too soon and may stall waiting for the bus. If this figure is too high, bus access may be granted but the bus may stall waiting on the design to start the access. Default latency in Vitis HLS is 64.

max_read_burst_length
: Specifies the maximum number of data values read during a burst transfer. Default value is 16.

num_read_outstanding
: Specifies how many read requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design: a FIFO of size `num_read_outstanding` * `max_read_burst_length` * `word_size`. Default value is 16.

max_write_burst_length
: Specifies the maximum number of data values written during a burst transfer. Default value is 16.

num_write_outstanding
: Specifies how many write requests can be made to the AXI4 bus, without a response, before the design stalls. This implies internal storage in the design: a FIFO of size `num_read_outstanding` * `max_read_burst_length` * `word_size`. Default value is 16.

The following INTERFACE pragma example can be used to help explain these options:

```
#pragma HLS interface m_axi port=input offset=slave bundle=gmem0
depth=1024*1024*16/(512/8) latency=100 num_read_outstanding=32 num_write_outstanding=32
max_read_burst_length=16 max_write_burst_length=16
```

The interface is specified as having a latency of 100. The HLS compiler seeks to schedule the request for burst access 100 clock cycles before the design is ready to access the AXI4 bus. To further improve bus efficiency, the options `num_write_outstanding` and `num_read_outstanding` ensure the design contains enough buffering to store up to 32 read

and/or write accesses. Each request will require its own buffer. This allows the design to continue processing until the bus requests are serviced. Finally, the options `max_read_burst_length` and `max_write_burst_length` ensure the maximum burst size is 16 and that the AXI4 interface does not hold the bus for longer than this. These options allow the behavior of the AXI4 interface to be optimized for the system in which it will operate. The efficiency of the operation does depend on these values being set accurately. The default values were specified as a conservative bound and may require changing depending on the memory access profile of the design being accelerated.

Table 10. Vitis HLS Controls

Vitis HLS Command	**Value**	**Description**
config_rtl - m_axi_conservative_mode	bool default=false	Delay M-AXI each write request until the associated write data are entirely available (typically, buffered into the adaptor or already emitted). This can slightly increase write latency but can resolve deadlock due to concurrent requests (read or write) on the memory subsystem.
config_interface - m_axi_latency	uint 0 is auto default=0 (flow_target=vivado) default=64 (flow_target=vitis)	Provide the scheduler with an expected latency for M-AXI accesses. Latency is the delay between a read request and the first read data, or between the last write data and the write response. Note that this number need not be exact, underestimation makes for a lower-latency schedule, but with longer dynamic stalls. The scheduler will account for the additional adapter latency and add a few cycles.
config_interface - m_axi_min_bitiwidth	uint default=8	Minimum bitwidth for M-AXI interfaces data channels. Must be a power-of-two between 8 and 1024. Note that this does not necessarily increase throughput if the actual accesses are smaller than the required interface.
config_interface - m_axi_max_bitwidth	uint default=1024	Minimum bitwidth for M-AXI interfaces data channels. Must be a power-of-two between 8 and 1024. Note that this does decrease throughput if the actual accesses are bigger than the required interface as they will be split into a multi-cycle burst of accesses.
config_interface - m_axi_max_widen_bitwidth	uint default=0 (flow_target=vivado) default=512 (flow_target=vitis)	Allow the tool to automatically widen bursts on M-AXI interfaces up to the chosen bitwidth. Must be a power-of-two between 8 and 1024. Note that burst widening requires strong alignment properties (in addition to burst).
config_interface - m_axi_auto_max_ports	bool default=false	If the option is false, all the M-AXI interfaces that are not explicitly bundled will be bundled into a single common interface, thus minimizing resource usage (single adaptor). If the option is true, all the M-AXI interfaces that are not explicitly bundled will be mapped into individual interfaces, thus increasing the resource usage (multiple adaptors).
config_interface - m_axi_alignment_byte_size	uint default=0 (flow_target=vivado) default=64 (flow_target=vitis)	Assume top function pointers that are mapped to M-AXI interfaces are at least aligned to the provided width in byte (power of two). This can help automatic burst widening. Warning: behavior will be incorrect if the pointer are not actually aligned at run-time.
config_interface - m_axi_num_read_outstanding	uint default=16	Default value for M-AXI num_read_outstanding interface parameter.
config_interface - m_axi_num_write_outstanding	uint default=16	Default value for M-AXI num_write_outstanding interface parameter.

Vitis HLS Command	Value	Description
config_interface -m_axi_max_read_burst_length	uint default=16	Default value for M-AXI max_read_burst_length interface parameter.
config_interface -m_axi_max_write_burst_length	uint default=16	Default value for M-AXI max_write_burst_length interface parameter.

Examples of Recommended Coding Styles

Vitis HLS issues INFO messages whenever it finds opportunities for bursting. It does not however issue messages if bursting was not possible as this would result in too many messages. The compiler will even report some statistics like burst length and burst bit widths when it can. If bursts of variable lengths are done, then the message will mention that bursts of variable lengths were inferred. These messages can be found quite early in the compiler log, `vitis_hls.log`. These messages are issued before the scheduling step.

Simple Read/Write Burst Inference

The following example is the standard way of reading and writing to the DDR and inferring a read and write burst. The Vitis HLS compiler will report the following burst inferences for the example below:

```
INFO: [HLS 214-115] Burst read of variable length and bit width 32 has been inferred on port 'gmem'
INFO: [HLS 214-115] Burst write of variable length and bit width 32 has been inferred on port 'gmem'
(./src/vadd.cpp:75:9).
```

The code for this example follows:

```
/****** BEGIN EXAMPLE *******/

#define DATA_SIZE 2048
// Define internal buffer max size
#define BURSTBUFFERSIZE 256

//TRIPCOUNT identifiers
const unsigned int c_min = 1;
const unsigned int c__max = BURSTBUFFERSIZE;
const unsigned int c_chunk_sz = DATA_SIZE;

extern "C" {
void vadd(int *a, int size, int inc_value) {
    // Map pointer a to AXI4-master interface for global memory access
#pragma HLS INTERFACE m_axi port=a  offset=slave bundle=gmem max_read_burst_length=256
max_write_burst_length=256
    // We also need to map a and return to a bundled axilite slave interface
#pragma HLS INTERFACE s_axilite port=a bundle=control
#pragma HLS INTERFACE s_axilite port=size bundle=control
#pragma HLS INTERFACE s_axilite port=inc_value bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control

    int burstbuffer[BURSTBUFFERSIZE];

    // Per iteration of this loop perform BURSTBUFFERSIZE vector addition
    for (int i = 0; i < size; i += BURSTBUFFERSIZE) {
#pragma HLS LOOP_TRIPCOUNT min=c_min*c_min max=c_chunk_sz*c_chunk_sz/(c_max*c_max)
        int chunk_size = BURSTBUFFERSIZE;
        //boundary checks
        if ((i + BURSTBUFFERSIZE) > size)
            chunk_size = size - i;

        // memcpy creates a burst access to memory
        // multiple calls of memcpy cannot be pipelined and will be scheduled sequentially
```

Accessing Row Data from a Two-Dimensional Array

The following is an example of reading/writing to/from a two dimensional array. Vitis HLS infers read and write bursts and issues the following messages:

```
INFO: [HLS 214-115] Burst read of length 256 and bit width 512 has been inferred on port 'gmem'
(./src/row_array_2d.cpp:43:5)
INFO: [HLS 214-115] Burst write of length 256 and bit width 512 has been inferred on port 'gmem'
(./src/row_array_2d.cpp:56:5)
```

The code for this example follows:

```
/****** BEGIN EXAMPLE *******/
// Parameters Description:
//         NUM_ROWS:            matrix height
//         WORD_PER_ROW:        number of words in a row
//         BLOCK_SIZE:          number of words in an array
#define NUM_ROWS   64
#define WORD_PER_ROW 64
#define BLOCK_SIZE (WORD_PER_ROW*NUM_ROWS)

// Default datatype is integer
typedef int DTYPE;
typedef hls::stream<DTYPE> my_data_fifo;

// Read data function: reads data from global memory
void read_data(DTYPE *inx, my_data_fifo &inFifo) {
read_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
    read_loop_jj:
        for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
           #pragma HLS PIPELINE II=1
            inFifo << inx[WORD_PER_ROW * i + jj];
            ;
        }
    }
}

// Write data function - writes results to global memory
void write_data(DTYPE *outx, my_data_fifo &outFifo) {
write_loop_i:
    for (int i = 0; i < NUM_ROWS; ++i) {
    write_loop_jj:
        for (int jj = 0; jj < WORD_PER_ROW; ++jj) {
           #pragma HLS PIPELINE II=1
            outFifo >> outx[WORD_PER_ROW * i + jj];
        }
```

Summary

Write code in such a way that bursting can be inferred. Ensure that none of the preconditions are violated.

Bursting does not mean that you will get all your data in one shot – it is about merging the requests together into one request, but the data will arrive sequentially, one after another.

Burst length of 16 is ideal, but even burst lengths of 8 are enough. Bigger bursts have more latency while shorter bursts can be pipelined. Do not confuse bursting with pipelining, but note that bursts can be pipelined with other bursts.

Bigger bursts have higher priority with the AXI interconnect. No dynamic arbitration is done inside the kernel.

You can have two `m_axi` ports connected to same DDR to model mutually exclusive access inside kernel, but the AXI interconnect outside the kernel will arbitrate competing requests.

One way to get around the out-of-order access restriction is to create your own buffer in BRAM, store the bursts in this buffer and then use this buffer to do out of order accesses. This is typically called a line buffer and is a common optimization used in video processing.

Adding RTL Blackbox Functions

The RTL blackbox enables the use of existing RTL IP in an HLS project. This lets you add RTL code to your C-code for synthesis of the project by Vitis HLS. The RTL IP can be used in a sequential, pipeline, or dataflow region.

Integrating RTL IP into a Vitis HLS project requires the following files:

- C function signature for the RTL code. This can be placed into a header (`.h`) file.
- Blackbox JSON description file as discussed in JSON File for RTL Blackbox.
- RTL IP files.

To use the RTL blackbox in an HLS project, use the following steps.

1. Call the C function signature from within your top-level function, or a sub-function in the Vitis HLS project.
2. Add the blackbox JSON description file to your HLS project using the **Add Files** command from the Vitis HLS GUI as discussed in Creating a New Vitis HLS Project (creatingnewvitishlsproject.html#thg1583443745171), or using the `add_files` command:

   ```
   add_files -blackbox my_file.json
   ```

   TIP: As explained in the next sectiom, the new RTL BlackBox wizard can help you generate the JSON file and add the RTL IP to your project.

3. Run the Vitis HLS design flow for simulation, synthesis, and co-simulation as usual.

Requirements and Limitations

RTL IP used in the RTL blackbox feature have the following requirements:

- Should be Verilog (.v) code.
- Must have a unique clock signal, and a unique active high reset signal.
- Must have a CE signal that is used to enable or stall the RTL IP.
- Must use the `ap_ctrl_chain` protocol as described in Block-Level I/O Protocols.

Within Vitis HLS, the RTL blackbox feature:

- Supports only C++.
- Cannot connect to top-level interface I/O signals.
- Cannot directly serve as the design-under-test (DUT).
- Does not support `struct` or `class` type interfaces.
- Supports the following interface protocols as described in JSON File for RTL Blackbox:

  hls::stream

  The RTL blackbox IP supports the `hls::stream` interface. When this data type is used in the C function, use a `FIFO` RTL port protocol for this argument in the RTL blackbox IP.

  Arrays

  The RTL blackbox IP supports RAM interface for arrays. For array arguments in the C function, use one of the following RTL port protocols for the corresponding argument in the RTL blackbox IP:
  - Single port RAM – RAM_1P
  - Dual port RAM – RAM_T2P

  Scalars and Input Pointers

  The RTL Blackbox IP supports C scalars and input pointers only in sequential and pipeline regions. They are not supported in a dataflow region. When these constructs are used in the C function, use `wire` port protocol in the RTL IP.

  Inout and Output Pointers

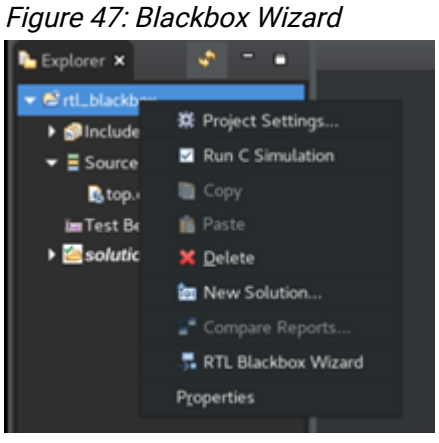
  The RTL blackbox IP supports inout and output pointers only in sequential and pipeline regions. They are not supported in a dataflow region. When these constructs are used in the C function, the RTL IP should use `ap_vld` for output pointers, and `ap_ovld` for inout pointers.

TIP: All other Vitis HLS design restrictions also apply when using RTL blackbox in your project.

Using the BlackBox Wizard

Navigate to the project, right click to open the Blackbox wizard as shown in the following figure:

Figure 47: Blackbox Wizard

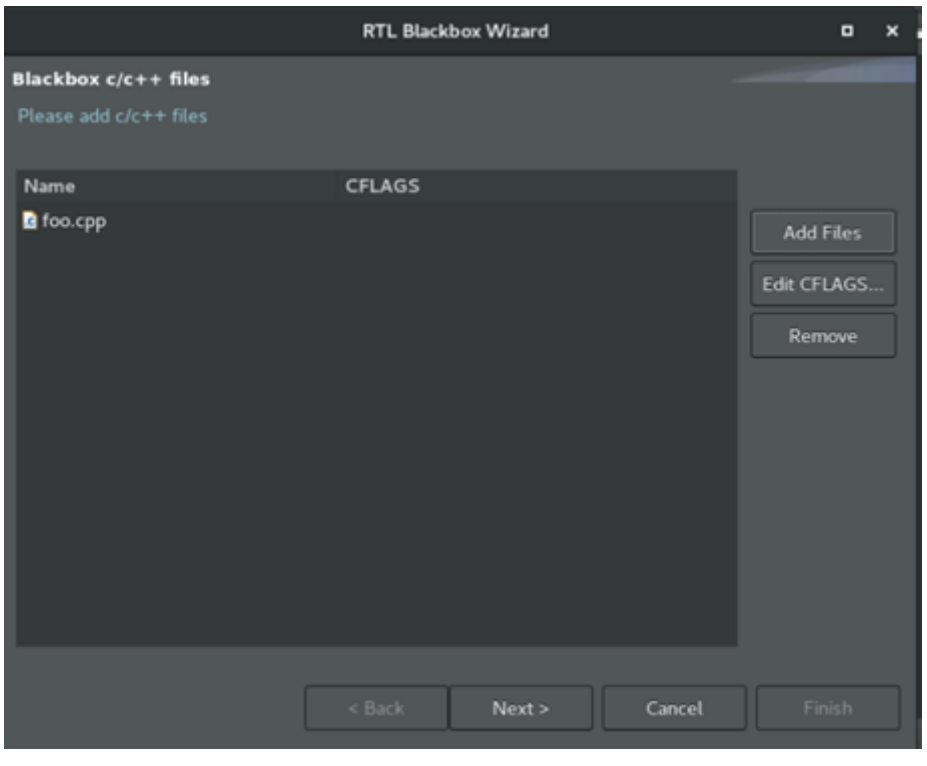


The Wizard is organized into pages that break down the process for creating a JSON file. To navigate between pages, click Next and select Back. Once the options are finalized the user can generate a JSON by clicking OK. Each of the following section describes each page and its input options

C++ Model and Header Files

In the Blackbox C/C++ Files dialog box, you provide the C++ files which form the functional model of the RTL IP. This C++ model is only used during C-simulation and C/RTL co-simulation. The RTL IP is combined with Vitis HLS results to form the output of synthesis.

Figure 48: Blackbox C/C++ Files



In this dialog box, you can do the following:

- Click **Add Files** to add files.
- Click **Edit CFLAGS** to provide a linker flag to the functional C model.
- Click **Next** to proceed.

The C File Wizard page lets you specify the values used for the C++ functional model of the RTL IP. The fields include:

C Function
: Specify the C++ function name of the RTL IP.

C Argument Name
: Specify the name(s) of the function arguments. These should relate to the ports on the IP.

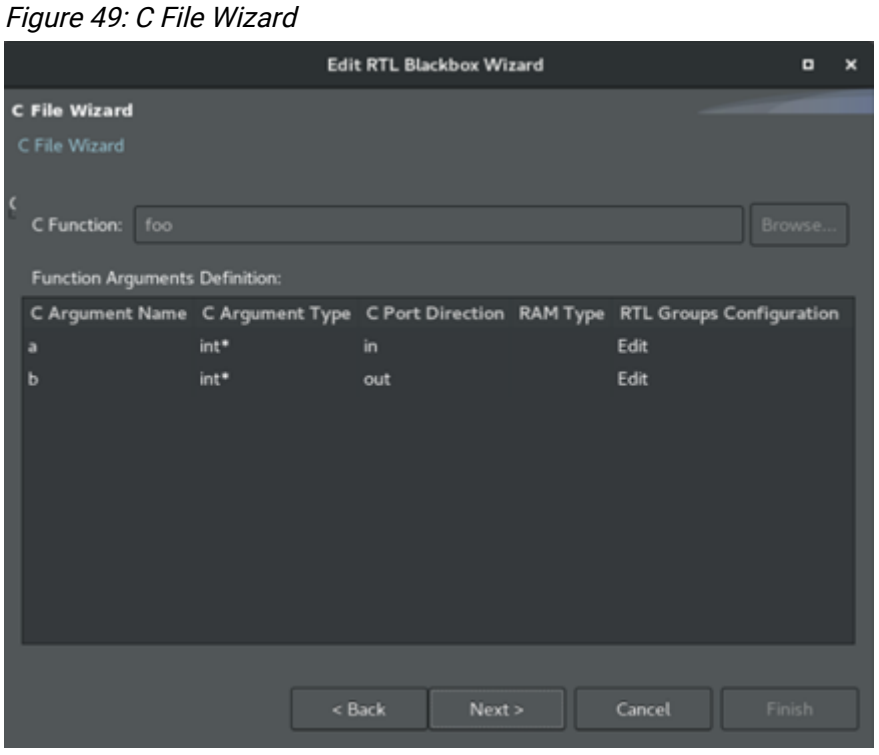
C Argument Type
: Specify the data type used for each argument.

C Port Direction
: Specify the port direction of the argument, corresponding to the port in the IP.

RAM Type
: Specify the RAM type used at the interface.

RTL Group Configuration
: Specifies the corresponding RTL signal name.

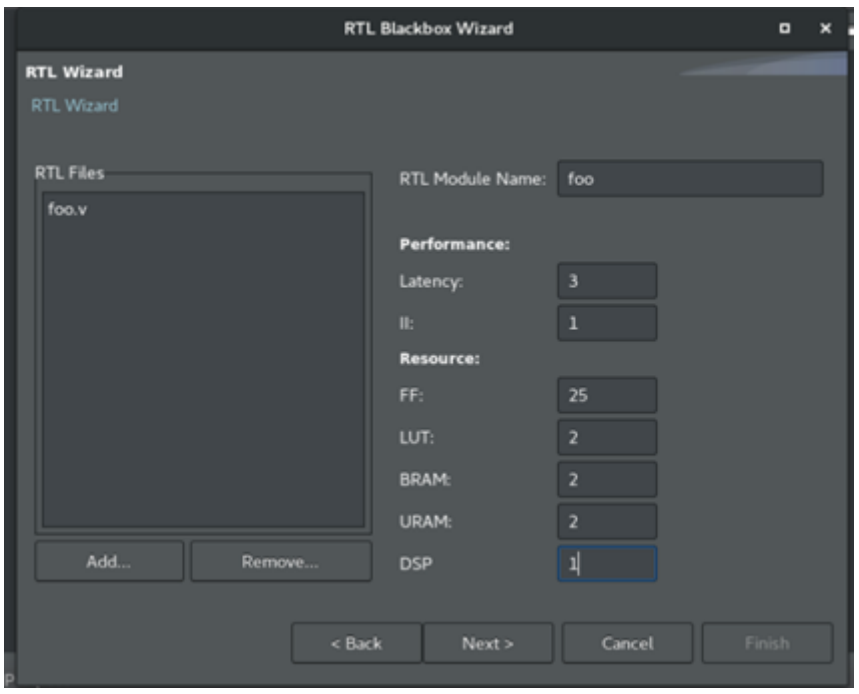
Figure 49: C File Wizard



Click **Next** to proceed.

RTL IP Definition

Figure 50: RTL Blackbox Wizard



The RTL Wizard page lets you define the RTL source for the IP. The fields to define include:

RTL Files

This option is used to add or remove the pre existing RTL IP files.

RTL Module Name

Specify the top level RTL IP module name in this field.

Performance

Specify performance targets for the IP.

Latency

Latency is the time required for the design to complete. Specify the Latency information in this field.

II

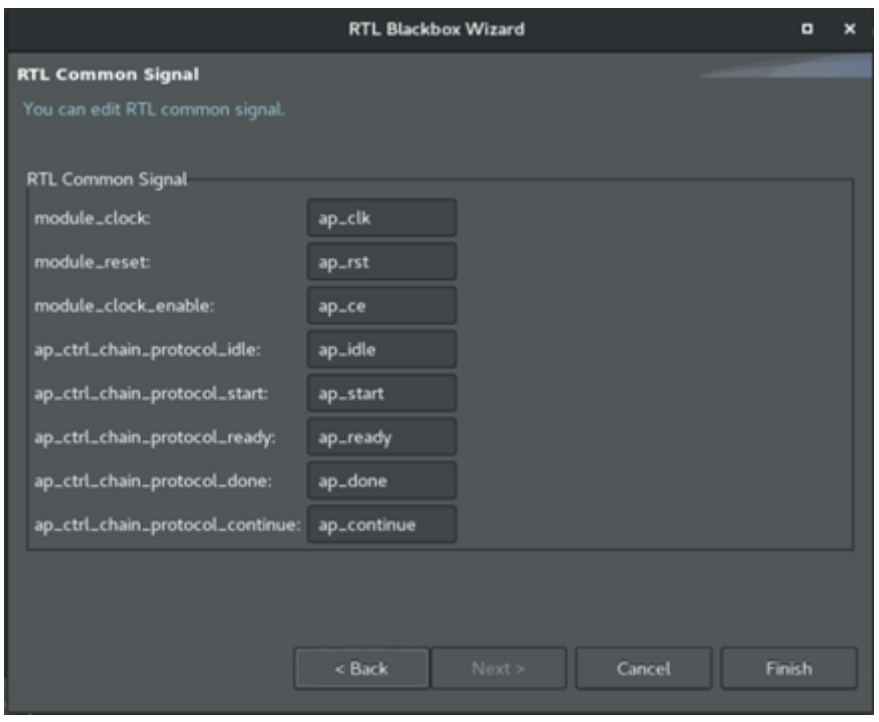
Define the target II (Initiation Interval). This is the number of clocks cycles before new input can be applied.

Resource

Specify the device resource utilization for the RTL IP. The resource information provided here will be combined with utilization from synthesis to report the overall design resource utilization. You should be able to extract this information from the Vivado Design Suite

Click **Next** to proceed to the RTL Common Signal page, as shown below.

Figure 51: RTL Common Signals



module_clock

Specify the name the of the clock used in the RTL IP.

module_reset

Specify the name of the reset signal used in the IP.

module_clock_enable

Specify the name of the clock enable signal in the IP.

ap_ctrl_chain_protocol_start

Specify the name of the block control start signal used in the IP.

ap_ctrl_chain_protocol_ready

Specify the name of the block control ready signal used in the IP.

ap_ctrl_chain_protocol_done

Specify the name of the block control done signal used in the IP.

ap_ctrl_chain_protocol_continue

Specify the name of the block control continue signal used in the RTL IP.

Click **Finish** to automatically generate a JSON file for the specified IP. This can be confirmed through the log message as shown below.

Log Message: `"[2019-08-29 16:51:10] RTL Blackbox Wizard Information: the "foo.json" file has been created in the rtl_blackbox/Source folder."`

The JSON file can be accessed through the Source file folder, and will be generated as described in the next section.

JSON File for RTL Blackbox

JSON File Format

The following table describes the JSON file format:

Table 11. JSON File Format

Item	Attribute	Description
c_function_name		The C++ function name for the blackbox. The **c_function name** must be consistent with the C function simulation model.
rtl_top_module_name		The RTL function name for the blackbox. The **rtl_top_module_name** must be consistent with the **c_function_name**.
c_files	c_file	Specifies the c file used for the blackbox module.
	cflag	Provides any compile option necessary for the corresponding c file.
rtl_files		Specifies the RTL files for the blackbox module.
c_parameters	c_name	Specifies the name of the argument used for the black box C++ function. Unused **c_parameters** should be deleted from the template.
	c_port_direction	The access direction for the corresponding c argument. **in** Read only by blackbox C++ function. **out** Write only by blackbox C++ function. **inout** Will both read and write by blackbox C++ function.
	RAM_type	Specifies the RAM type to use if the corresponding C argument uses the RTL 'RAM protocol. Two type of RAM are used: **RAM_1P** For 1 port RAM module **RAM_T2P** For 2 port RAM module Omit this attribute when the corresponding C argument is not using RTL 'RAM' protocol.

Item	Attribute	Description
	rtl_ports	Specifies the RTL port protocol signals for the corresponding c argument (`c_name`). Every **c_parameter** should be associated with an **rtl_port**. Five type of RTL port protocols are used. Refer to the **RTL Port Protocols** table for additional details. **wire** An argument can be mapped to `wire` if it is a scalar or pointer with 'input' direction. **ap_vld** An argument can be mapped to `ap_vld` if it uses pointer with 'out' direction. **ap_ovld** An argument can be mapped to `ap_ovld` if it use a pointer with an 'inout' direction. **FIFO** An argument can be mapped to `FIFO` if it uses the `hls::stream` datatype. **RAM** An argument can be mapped to `RAM` if it uses an array type. The array type supports inout directions. The specified RTL port protocols have associated control signals, which also need to be specified in the JSON file.
c_return	c_port_direction	It must be `out` .
	rtl_ports	Specifies the corresponding RTL port name used in the RTL blackbox IP.
rtl_common_signal	module_clock	The unique clock signal for RTL blackbox module.
	module_reset	Specifies the reset signal for RTL blackbox module. The reset signal must be active high or positive valid.
	module_clock_enable	Specifies the clock enable signal for the RTL blackbox module. The enable signal must be active high or positive valid.
	ap_ctrl_chain_protocol_idle	The ap_idle signal in the `ap_ctrl_chain` protocol for the RTL blackbox module.
	ap_ctrl_chain_protocol_start	The ap_start signal in the `ap_ctrl_chain` protocol for the RTL blackbox module.
	ap_ctrl_chain_protocol_ready	The ap_ready signal in the `ap_ctrl_chain` protocol for the RTL blackbox IP.
	ap_ctrl_chain_protocol_done	The 'ap_done' signal in the ap_ctrl_chain protocol for blackbox RTL module.
	ap_ctrl_chain_protocol_continue	The ap_continue signal in the `ap_ctrl_chain` protocol for RTL blackbox module.
rtl_performance	latency	Specifies the Latency of the RTL backbox module. It must be a non-negative integer value. For Combinatorial RTL IP specify `0` , otherwise specify the exact latency of the RTL module.
	II	Number of clock cycles before the function can accept new input data. It must be non-negative integer value. `0` means the blackbox can not be pipelined. Otherwise, it means the blackbox module is pipelined..
rtl_resource_usage	FF	Specifies the register utilization for the RTL blackbox module.
	LUT	Specifies the LUT utilization for the RTL blackbox module.

Item	Attribute	Description
	BRAM	Specifies the block RAM utilization for the RTL blackbox module.
	URAM	Specifies the URAM utilization for the RTL blackbox module.
	DSP	Specifies the DSP utilization for the RTL blackbox module.

Table 12. RTL Port Protocols

RTL Port Protocol	RAM Type	C Port Direction	Attribute	User-Defined Name	Notes
wire		in	data_read_in	Specifies a user defined name used in the RTL blackbox IP. As an example for wire, if the RTL port name is "flag" then the JSON FILE format is "data_read-in" : "flag"	
ap_vld		out	data_write_out		
			data_write_valid		
ap_ovld		inout	data_read_in		
			data_write_out		
			data_write_valid		
FIFO		in	FIFO_empty_flag		Must be negative valid.
			FIFO_read_enable		
			FIFO_data_read_in		
		out	FIFO_full_flag		Must be negative valid.
			FIFO_write_enable		
			FIFO_data_write_out		
RAM	RAM_1P	in	RAM_address		
			RAM_clock_enable		
			RAM_data_read_in		
		out	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
		inout	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
			RAM_data_read_in		
RAM	RAM_T2P	in	RAM_address	Specifies a user defined name used in the RTL blackbox IP. As an example for wire, if the RTL port name is "flag" then the JSON FILE format is "data_read-in" : "flag"	Signals with _snd belong to the second port of the RAM. Signals without _snd belong to the first port.
			RAM_clock_enable		
			RAM_data_read_in		
			RAM_address_snd		
			RAM_clock_enable_snd		
			RAM_data_read_in_snd		

RTL Port Protocol	RAM Type	C Port Direction	Attribute	User-Defined Name	Notes
		out	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
			RAM_address_snd		
			RAM_clock_enable_snd		
			RAM_write_enable_snd		
			RAM_data_write_out_snd		
		inout	RAM_address		
			RAM_clock_enable		
			RAM_write_enable		
			RAM_data_write_out		
			RAM_data_read_in		
			RAM_address_snd		
			RAM_clock_enable_snd		
			RAM_write_enable_snd		
			RAM_data_write_out_snd		
			RAM_data_read_in_snd		

Note: The behavioral C-function model for the RTL blackbox must also adhere to the recommended HLS coding styles.

JSON File Example

This section provides details on manually writing the JSON file required for the RTL blackbox. The following is an example of a JSON file:

```
{
"c_function_name" : "foo",
"rtl_top_module_name" : "foo",
"c_files" :
      [
            {
              "c_file" : "../../a/top.cpp",
              "cflag" : ""
            },
            {
              "c_file" : "xx.cpp",
              "cflag" : "-D KF"
            }
           ],
"rtl_files" : [
                "../../foo.v",
                "xx.v"
              ],
"c_parameters" : [{
                   "c_name" : "a",
                   "c_port_direction" : "in",
                   "rtl_ports" : {
                                  "data_read_in" : "a"
                                 }
                  },
                  {
                   "c_name" : "b",
                   "c_port_direction" : "in",
                   "rtl_ports" : {
                                  "data_read_in" : "b"
                                 }
                  },
                  {
                   "c_name" : "c",
                   "c_port_direction" : "out",
```