Data Types

IMPORTANT! *Due to library dependencies that are only compiled for 64-bit systems, Vitis HLS does not support 32-bit builds. Due to this, usage of* `-m32` *flag is not allowed and will cause an error.*

The data types used in a C/C++ function compiled into an executable impact the accuracy of the result and the memory requirements, and can impact the performance. A 32-bit integer int data type can hold more data and therefore provide more precision than an 8-bit char type, but it requires more storage. Similarly, when the C/C++ function is to be synthesized to an RTL implementation, the types impact the precision, the area, and the performance of the RTL design. The data types used for variables determine the size of the operators required and therefore the area and performance of the RTL.

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

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

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

The C/C++ standard dictates that type `(unsigned)long` is implemented as 64 bits on 64-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.

AMD 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.*

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

Standard Types

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

```
#include "types_standard.h"

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

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

}
```

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

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

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

#define N 9

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

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

void types_standard(din_A inA,din_B inB,din_C inC,din_D inD,dout_1

*out1,dout_2 *out2,dout_3 *out3,dout_4 *out4);
```

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

- The multiplier used to calculate result `out1` is a 24-bit multiplier. An 8-bit `char` type multiplied by a 16-bit `short` type requires a 24-bit multiplier. The result is sign-extended to 32-bit to match the output port width.
- The adder used for `out2` is 8-bit. Because the output is an 8-bit `unsigned char` type, only the bottom 8-bits of `inB` (a 16-bit `short`) are added to 8-bit `char` type `inA`.
- For output `out3` (32-bit exact width type), 8-bit `char` type `inA` is sign-extended to 32-bit value and a 32-bit division operation is performed with the 32-bit (`int` type) `inC` input.
- A 64-bit modulus operation is performed using the 64-bit `long long` type `inD` and 8-bit `char` type `inA` sign-extended to 64-bit, to create a 64-bit output result `out4`.

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

Floats and Doubles

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

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

RECOMMENDED: *When using floating-point data types, AMD highly recommends that you review Floating-Point Design with Vivado HLS (XAPP599). Also refer to Vitis-HLS-Introductory-Examples/Modeling/using_float_and_double on Github for an example of using floating and double data types.*

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 AMD IP catalog cores.

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

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

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

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

The above code results in the following hardware:

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

Using a `sqrtf()` function:

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

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

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

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

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

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

Floating-Point Accumulator and MAC

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

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

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

- Low-precision accumulation is suitable for high-throughput low-precision floating point accumulation and multiply-accumulation, this mode is only available in non-Versal devices.
  - It uses an integer accumulator with a pre-scaler and a post-scaler (to convert input and output to single-precision or double-precision floating point).
    - It uses a 60 bit and 100 bit accumulator for single and double precision inputs respectively.
    - It can cause cosim mismatches due to insufficient precision with respect to C++ simulation
  - It can always be pipelined with an II=1 without source code changes
  - It uses approximately 3X the resources of standard-precision floating point accumulation, which achieves an II that is typically between 3 and 5, depending on clock frequency and target device.

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

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

- Standard-precision accumulation and multiply-add is suitable for most uses of floating-point, and is available on Versal and non-Versal devices.
  - It always uses a true floating-point accumulator
  - It can be pipelined with an II=1 on Versal devices.
  - It can be pipelined with an II that is typically between 3 and 5 (depending on clock frequency and target device) on non-Versal devices. The standard precision mode is more efficient on Versal devices than on non-Versal devices.
- High-precision fused multiply-add is suitable for high-precision applications and is available on Versal devices.

- It uses one extra bit of precision
- It always uses a single fused multiply-add, with a single rounding at the end, although it uses more resources than the unfused multiply-add
- It can cause cosim mismatches due to the extra precision with respect to C++ simulation

Composite Data Types

HLS supports composite data types for synthesis:

- Structs
- Enumerated Types
- Unions

Structs

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

IMPORTANT! *Structs used as arguments to the top-level function are aggregated by default as described in Structs in the Interface.*

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

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

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

```
config_unroll -tripcount_threshold 16
```

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

Structs in the Interface

Structs in the interface are kept 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. You can disaggregate structs in the interface by using the DISAGGREGATE pragma or directive. When a struct contains one or more `hls::stream` objects Vitis HLS will automatically disaggregate the struct as described below in *Structs in the Interface with `hls::stream` Elements*.

> **IMPORTANT!** *Disaggregating a struct in the interface is not supported in the Vitis kernel flow because the Vitis tool cannot map a single C-argument to multiple RTL ports. When disaggregating a struct in the interface, either manually or automatically, Vitis HLS will build and export the Vitis kernel output (`.xo`), but that output will result in an error when used with the `v++` command. To support the Vitis Kernel flow you must manually break the struct into its constituent elements, and define any `hls::stream` objects as using an AXIS interface.*

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

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

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

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

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

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

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

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

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

Structs on the Interface with hls::stream Elements

User-defined structs on the interface containing `hls::stream` elements are automatically disaggregated by Vitis HLS. This disaggregated struct is supported in the Vivado IP flow, and the exported IP will work as expected. However, this disaggregated struct is not supported for the Vitis Kernel flow, and the exported kernel (`.xo`) will cause an error when used with the `v++ --link` command. To support the Vitis Kernel flow you must manually break the struct into its constituent elements, and define the `hls::stream` object as using an AXIS interface.

If you have a struct that is disaggregated automatically, Vitis HLS applies any INTERFACE pragmas to the individual elements of the disaggregated struct. If there is only one INTERFACE pragma specified for the struct, it is applied to each element of the struct. If you provide an INTERFACE pragma for each element of the disaggregated struct, it is applied as expected.

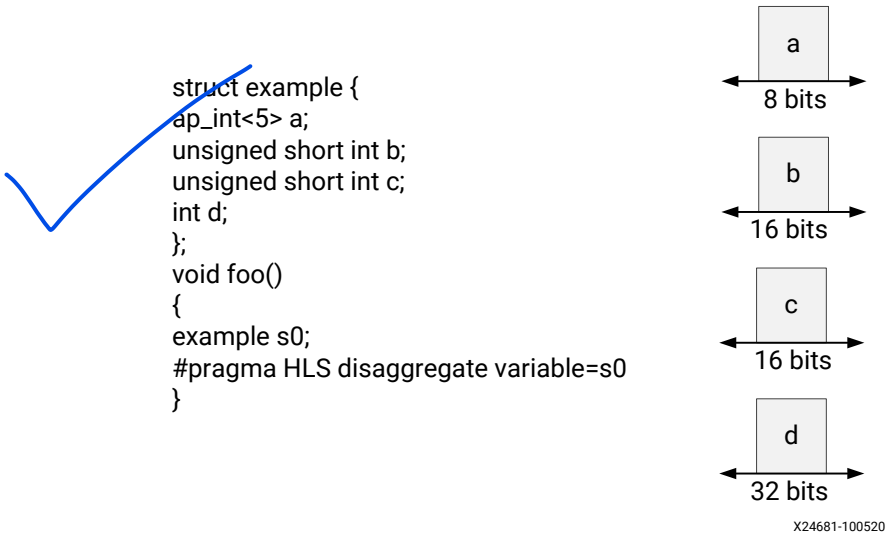
Struct Padding and Alignment

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

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

TIP: *You can use the AGGREGATE pragma or directive to prevent the default disaggregation of structs in the code.*

Figure 31: **Disaggregated Struct**



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

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

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

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

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

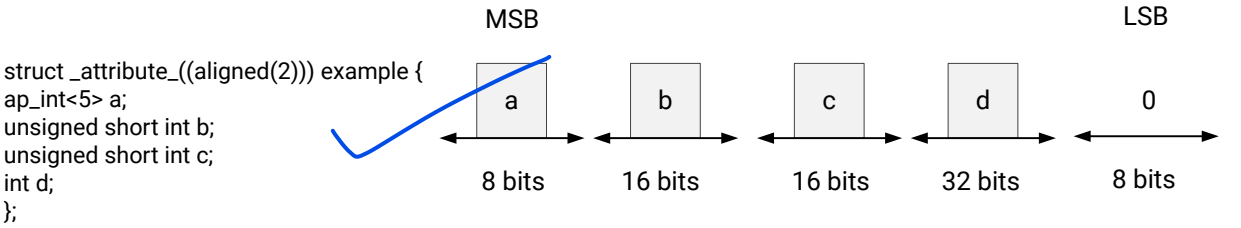
  Data types in struct with custom data widths, such as `ap_int`, are allocated with sizes which are powers of 2. Vitis HLS adds padding bits for aligning the size of the data type to a power of 2.

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

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

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

Figure 32: **Aligned Struct Implementation**



X24682-102220

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

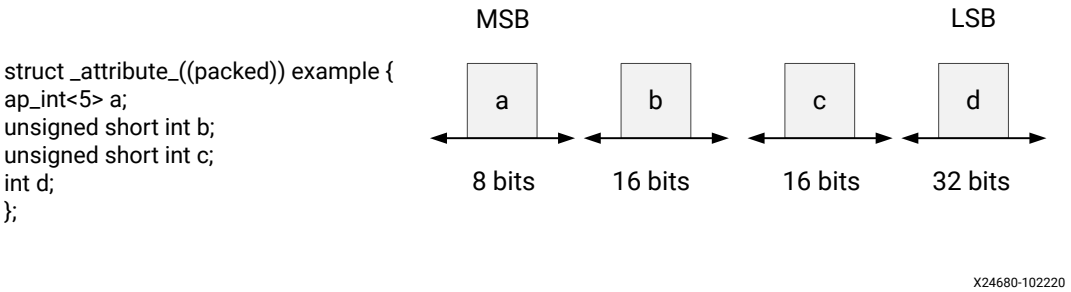
```
struct data_t {
  short varA;
  int varB;
  short varC;
};
```

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

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

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

Figure 33: **Packed Struct Implementation**



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

C++ Classes and Templates

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

```
#include "cpp_FIR.h"

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

 cout << fir1;

 return fir1(x);
 }
```

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

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

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

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

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

#define N 85

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

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

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

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

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

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

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

data_t cpp_FIR(data_t x);
```

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

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

```
#include "cpp_FIR.h"

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


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

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

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

 }
 result.close();

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

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

C++ Test Bench for `cpp_FIR`

To apply directives to objects defined in a class:

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

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

Global Variables and Classes

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

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

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

// Use k on line 73 static int k;

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

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

     }
};

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

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

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

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

col);

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


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

 }
 sample_count++;
}
```

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

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

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

Templates

Vitis HLS supports the use of templates in C++ for synthesis. Vitis HLS does not support templates for the top-level function. Refer to Vitis-HLS-Introductory-Examples/Modeling/using_C++_templates on Github for an example of these concepts.

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

Using Templates to Create Unique Instances

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

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

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

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

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

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

Using Templates for Recursion

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

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

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

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

```
        return b;
 }
};

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

Enumerated Types

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

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

```
#include <stdio.h>

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

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

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

typedef   signed int mad_fixed_t;

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

 unsigned long long bitrate;
 unsigned int samplerate;

 unsigned short crc_check;
 unsigned short crc_target;

 int flags;
 int private_bits;

} header_t;

typedef struct mad_frame {
 header_t header;
```

```
 int options;
 mad_fixed_t sbsample[2][36][32];
} frame_t;

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


void types_composite(frame_t *frame);
```

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

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

```
#include "types_composite.h"

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

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

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

Unions

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

```
#include "types_union.h"

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

```
 unsigned long long one, exp;

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

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

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

Vitis HLS does *not* support the following:

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

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

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

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

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

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

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

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

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

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

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

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

Type Qualifiers

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

Volatile

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

***Note*:** Accesses to/from volatile variables is preserved. This means:

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

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

Statics

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

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

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

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

Const

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

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

ROM Optimization

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

```
#include "array_ROM.h"

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

 for (i = 0; i < 256; i++) {
```

```
 lookup_table[i] = 256 * (i - 128);
 }

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

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

Arbitrary Precision (AP) Data Types

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

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

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

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

AP Example

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

Using arbitrary precision data types in this design, you can specify the exact bit-sizes needed in your code prior to synthesis, simulate the updated code, and verify the results prior to synthesis. Refer to Vitis-HLS-Introductory-Examples/Modeling on Github for examples of using arbitrary precision and fixed point ap data types.

Advantages of AP Data Types

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

The following code performs some basic arithmetic operations:

```
#include "types.h"

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

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

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

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

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

The design gives the following results after synthesis:

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

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |   66|   66|   67|   67|   none  |
    +-----+-----+-----+-----+---------+
* Summary:
+-----------------+---------+-------+--------+--------+
|       Name      | BRAM_18K| DSP48E|   FF   |   LUT  |
```

```
+-----------------+---------+-------+--------+--------+
|Expression       |        -|      -|       0|      17|
|FIFO             |        -|      -|       -|       -|
|Instance         |        -|      1|   17920|   17152|
|Memory           |        -|      -|       -|       -|
|Multiplexer      |        -|      -|       -|       -|
|Register         |        -|      -|       7|       -|
+-----------------+---------+-------+--------+--------+
|Total            |        0|      1|   17927|   17169|
+-----------------+---------+-------+--------+--------+
|Available        |      650|    600|  202800|  101400|
+-----------------+---------+-------+--------+--------+
|Utilization (%)  |        0|     ~0|       8|      16|
+-----------------+---------+-------+--------+--------+
```

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

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

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

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

+ Latency (clock cycles):
    * Summary:
    +-----+-----+-----+-----+---------+
    |  Latency  |  Interval | Pipeline|
    | min | max | min | max |   Type  |
    +-----+-----+-----+-----+---------+
    |   35|   35|   36|   36|   none  |
    +-----+-----+-----+-----+---------+
* Summary:
+-----------------+---------+-------+--------+--------+
|       Name      | BRAM_18K| DSP48E|   FF   |   LUT  |
+-----------------+---------+-------+--------+--------+
|Expression       |        -|      -|       0|      13|
|FIFO             |        -|      -|       -|       -|
|Instance         |        -|      1|    4764|    4560|
|Memory           |        -|      -|       -|       -|
|Multiplexer      |        -|      -|       -|       -|
|Register         |        -|      -|       6|       -|
+-----------------+---------+-------+--------+--------+
|Total            |        0|      1|    4770|    4573|
```

```
+-----------------+---------+-------+--------+--------+
|Available        |      650|    600|  202800|  101400|
+-----------------+---------+-------+--------+--------+
|Utilization (%)  |        0|     ~0|       2|       4|
+-----------------+---------+-------+--------+--------+
```

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

Overview of Arbitrary Precision Integer Data Types

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

Table 2: **Arbitrary Precision Data Types**

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

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

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

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

```
#include "ap_int.h"

void foo_top () {

ap_int<9>  var1;            // 9-bit
ap_uint<10>  var2;            // 10-bit unsigned
```

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

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

The following is an example of overriding `AP_INT_MAX_W`:

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

ap_int<4096> very_wide_var;
```

Overview of Arbitrary Precision Fixed-Point Data Types

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

```
#include <ap_fixed.h>
...
ap_fixed<18,6,AP_RND> t1 = 1.5; // internally represented as
0b00'0001.1000'0000'0000 (0x01800)
ap_fixed<18.6,AP_RND> t2 = -1.5; // 0b11'1110.1000'0000'0000 (0x3e800)
...
```

When performing calculations where the variables have different number of bits or different precision, the binary point is automatically aligned. For example, when performing division with fixed-point type variables of different sizes, the fraction of the quotient is no greater than that of the dividend. To preserve the fractional part of the quotient you can cast the result to the new variable width before assignment.

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

Table 3: **Fixed-Point Identifier Summary** *(cont'd)*

<table>
<tr><th>Identifier</th><th colspan="2">Description</th></tr>
<tr><td>I</td><td colspan="2">The number of bits used to represent the integer value, that is, the number of integer bits to the left of the binary point. When this value is negative, it represents the number of implicit sign bits (for signed representation), or the number of implicit zero bits (for unsigned representation) to the right of the binary point. For example,

<pre>ap_fixed<2, 0> a = -0.5;   // a can be -0.5,

ap_ufixed<1, 0> x = 0.5;   // 1-bit representation. x can be 0 or 0.5
ap_ufixed<1, -1> y = 0.25; // 1-bit representation. y can be 0 or 0.25
const ap_fixed<1, -7> z = 1.0/256; // 1-bit representation for z = 2^-8</pre></td></tr>
<tr><td rowspan="10">Q</td><td colspan="2">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.</td></tr>
<tr><td>ap_fixed Types</td><td>Description</td></tr>
<tr><td>AP_RND</td><td>Round to plus infinity</td></tr>
<tr><td>AP_RND_ZERO</td><td>Round to zero</td></tr>
<tr><td>AP_RND_MIN_INF</td><td>Round to minus infinity</td></tr>
<tr><td>AP_RND_INF</td><td>Round to infinity</td></tr>
<tr><td>AP_RND_CONV</td><td>Convergent rounding</td></tr>
<tr><td>AP_TRN</td><td>Truncation to minus infinity (default)</td></tr>
<tr><td>AP_TRN_ZERO</td><td>Truncation to zero</td></tr>
<tr><td rowspan="8">O</td><td colspan="2">Overflow mode: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result.</td></tr>
<tr><td>ap_fixed Types</td><td>Description</td></tr>
<tr><td>AP_SAT[1]</td><td>Saturation</td></tr>
<tr><td>AP_SAT_ZERO[1]</td><td>Saturation to zero</td></tr>
<tr><td>AP_SAT_SYM[1]</td><td>Symmetrical saturation</td></tr>
<tr><td>AP_WRAP</td><td>Wrap around (default)</td></tr>
<tr><td>AP_WRAP_SM</td><td>Sign magnitude wrap around</td></tr>
<tr><td>N</td><td colspan="2">This defines the number of saturation bits in overflow wrap modes.</td></tr>
</table>

Notes:

1. Using the AP_SAT* modes can result in higher resource usage as extra logic will be needed to perform saturation and this extra cost can be as high as 20% additional LUT usage.
2. Fixed-point math functions from the `hls_math` library do not support the `ap_[u]fixed` template parameters Q,O, and N, for quantization mode, overflow mode, and the number of saturation bits, respectively. The quantization and overflow modes are only effective when an `ap_[u]fixed` variable is on the left hand of assignment or being initialized, but not during the calculation.

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

IMPORTANT! *ROM Synthesis can take a long time when using `ap_[u]fixed`. Changing it to `int` results in a quicker synthesis. For example:*

```
static ap_fixed<32,0> a[32][depth] =
```

Can be changed to:

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

Global Variables

Global variables can be freely used in the code and are fully synthesizable. However, global variables can not be inferred as arguments to the top-level function, but must instead be explicitly specified as arguments for ports in the RTL design.

The following code example shows the default synthesis behavior of global variables. It uses three global variables (`idx`, `Ain` and `Aout`). 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`.

IMPORTANT! *Access to the global variables `Ain` and `Aout` must be explicitly listed in the argument list.*

```
#include "top.h"

void top(const int idx, const int Ain[N], int Aout[Nhalf]) {

    int Aint[N];

    // Move elements in the input array

    ILOOP: for (int i = 0; i < N; i++) {

        int iadj = (i + idx) % N;

        Aint[i] = Ain[i] + Ain[iadj];

    } // end ILOOP

    // sum the 1st and 2nd halves
    OLOOP: for (int i = 0; i < Nhalf; i++) {

        Aout[i] = (Aint[i] + Aint[Nhalf + i]);

    } // end OLOOP

} // end top()
```

Pointers

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

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

***Note*:** Pointer to pointer is not supported.

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

```
#include "pointer_multi.h"

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

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

 return ptr[pos];
}
```

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

```
#include "pointer_double.h"

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

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

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

```
 data_t i;

 ptrFlag = flag;

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

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

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

```
#include "pointer_array.h"

data_t A[N][10];

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

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

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

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

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

 return sum1;
}
```

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

```
#define N 1024

typedef int data_t;
typedef char dint_t;

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

 // Sum from the indexed value as a different type
```

```
 for (i = 0; i < 4*(N/10); ++i) {
   result += *ptr;
   ptr+=1;
 }
 return result;
}
```

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

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

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

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

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

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

Pointers on the Interface

Pointers can be used as arguments to the top-level function. It is important to understand how pointers are implemented during synthesis, because they can sometimes cause issues in achieving the desired RTL interface and design after synthesis. Refer to Vitis-HLS-Introductory-Examples/Modeling/Pointers on Github for examples of some of the following concepts.

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 is shown in the following code example.

```
#include "pointer_basic.h"

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

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

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

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

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

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

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

Pointer Arithmetic

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

```
#include "pointer_arith.h"

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

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

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

```
#include "pointer_arith.h"

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

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

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

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

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

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

When simulated, this results in the following output:

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

The pointer arithmetic can access the pointer data out of sequence. On the other hand, wire, handshake, or FIFO interfaces can only access data in 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 starts reading from index 1 (`i` starts at 0, 0+1=1). This is the second element from array `d[5]` in the test bench.

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

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

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

```
#include "array_arith.h"

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

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

Multi-Access Pointers on the Interface

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

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

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

```
#include "pointer_stream_bad.h"

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

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

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

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

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

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

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

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

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

```
#include "pointer_stream_better.h"

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

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

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

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

Understanding Volatile Data

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

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

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

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

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

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

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

```
Din Dout
0   0
1   4
2   8
3   12
```

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

The `volatile` qualifier:

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

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

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

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

Modeling Streaming Data Interfaces

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

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

There are several possible approaches:

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

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

```
#include "pointer_stream_good.h"

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

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

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

```
#include "pointer_stream_good.h"

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

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

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

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

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

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

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

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

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

```
Din Dout
0   1
1   6
2
3
```

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

Multi-Access Pointers and RTL Simulation

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

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

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

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

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

Vector Data Types

The vector data type is provided to easily model and synthesize single instruction multiple data (SIMD) type operations. Many operators are overloaded to provide SIMD behavior for vector types. The AMD Vitis™ HLS library provides the reference implementation for the `hls::vector<T, N>` type which represent a single-instruction multiple-data (SIMD) vector, as defined below.

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

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

```
typedef  hls::vector<int, 8> t_int8Vec;
t_int8Vec intVectorA, intVectorB;
.
.
.
void processVecStream(hls::stream<t_int8Vec>
&inVecStream1,hls::stream<t_int8Vec> &inVecStream2, hls::stream<int8Vec>
&outVecStream)
{
    for(int i=0;i<32;i++)
    {
        #pragma HLS pipeline II=1
        t_int8Vec aVec = inVecStream1.read();
        t_int8Vec bBec = inVecStream2.read();
        //performs a vector operation on 8 integers in parallel
        t_int8Vec cVec = aVec * bVec;
        outVecStream.write(cVec);
    }
}
```

Refer to HLS Vector Library for additional information. Refer to Vitis-HLS-Introductory-Examples/Modeling/using_vectors on Github for an example.

Bit-Width Propagation

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 lower 24 bits are used for the result.

```
#include "ap_int.h"

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

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

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

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

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

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

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

Using arbitrary precision types for the two function inputs is enough to ensure Vitis HLS creates a design using a 24-bit multiplier. The 12-bit types are propagated through the design. AMD recommends that you correctly size the arguments of all functions in the hierarchy so that there is no need to size local variables.

In general, when variables are driven directly from the function interface, especially from the top-level function interface, variables 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.

Chapter 7

Unsupported C/C++ Constructs

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

To be synthesized:

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

System Calls

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

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

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

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

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

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

```
#include "hier_func4.h"

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

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

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

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

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

Dynamic Memory Usage

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

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

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

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

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

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

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

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

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

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

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

 return *out_accum;
}
```

Because the coding changes here impact the functionality of the design, AMD does not recommend using the `__SYNTHESIS__` macro. AMD recommends performing the following steps:

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

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

Pointer Limitations

General Pointer Casting

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

Pointer Arrays

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

Function Pointers

Function pointers are not supported.

Note: Pointer to pointer is not supported.

Recursive Functions

Recursive functions cannot be synthesized. This applies to functions that can form multiple recursions:

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

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

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

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

CAUTION! *Virtual Functions are not supported.*

Standard Template Libraries

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

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

Undefined Behaviors

The C/C++ undefined behaviors may lead to a different behavior in simulation and synthesis. An example of this behavior is shown below:

```
for (int i=0; i<N; i++) {
  int val; // uninitialized value
  if (i==0) val=0;
  else if (cond) val=1;
  // val may have indeterminate value here
  A[i] = val; // undefined behavior
  val++; // dead code
}
```

In the above example you should not expect that A[i] gets the value of val from the previous loop iteration if neither `i==0`, nor `(cond)` are true. You should even not expect that the increment (`val++`) will happen. The same is true for scalars values obtained after complete partition.

For such C/C++ undefined behavior situations, the behavior between GCC and Vitis HLS when compiling code is likely to be different, which will lead to a mismatch during RTL/Co-simulation. This is because in GCC, compiled for CPU, `val` is often left in the same register or in the same stack location across loop iterations, and therefore the behavior is that the value of `val` is retained between loop iterations.

The solution is either to initialize val at each iteration (if this is the expected behavior) or to move the declaration of `val` above the loop, as high as necessary, so that its lifetime scope matches the intent reuse. You should not expect that the compiler will infer a specific defined RTL behavior from an undefined C/C++ behavior.

Virtual Functions and Pointers

Not supported.