



# C/C++ Coding Styles

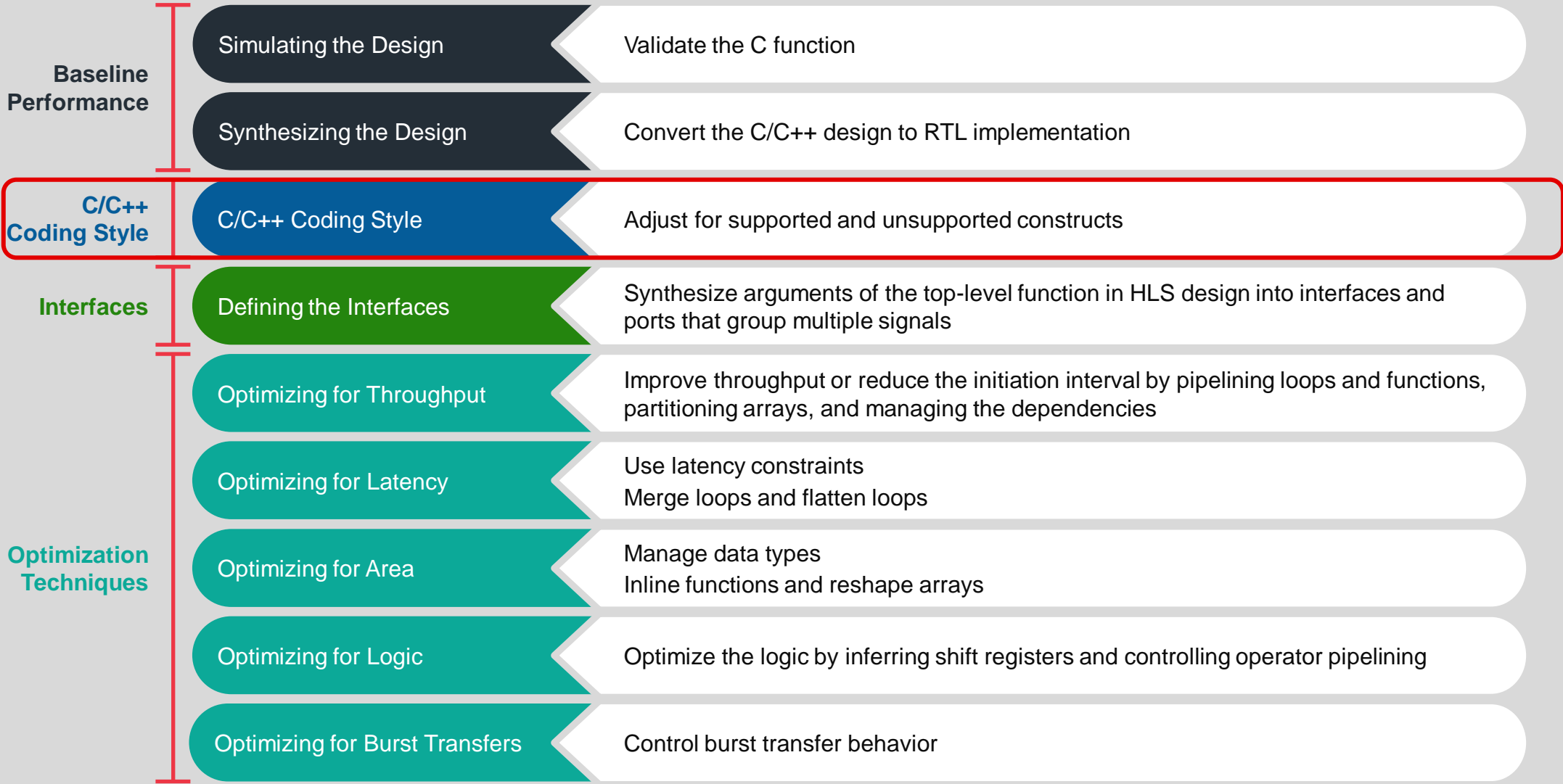
# Objectives

- ▶ After completing this module, you will be able to:
  - Describe the support offered by data types of C, C++
  - Define bit-accurate operators using arbitrary precision types
  - Identify advantages and pitfalls of using arbitrary precision
  - List various supported quantization and overflow modes
  - Describe the modeling issues present when using pointers

# Outline

- ▶ Vitis HLS Support
- ▶ Data Types and Bit-Accuracy
- ▶ Using Pointers with Limitation
- ▶ Stream
- ▶ Summary

# HLS Design Methodology



# Vitis HLS Support

# Comprehensive C Support

- ▶ A Complete C Validation & Verification Environment
  - Vitis HLS supports complete bit-accurate validation of the C model
  - Vitis HLS provides a productive C-RTL co-simulation verification solution
- ▶ Vitis HLS supports C, C++ and OpenCL API C kernel
  - Functions can be written in C and C++ 11/C++ 14
  - Wide support for coding constructs in C and C++ 11/C++ 14
- ▶ Modeling with bit-accuracy
  - Supports arbitrary precision types for all input languages
  - Allowing the exact bit-widths to be modeled and synthesized
- ▶ Floating point support
  - Support for the use of float and double in the code

# Unsupported Constructs: Overview

- System calls
  - System calls cannot be synthesized : performing some tasks on the OS in which the C program is running
  - Vitis HLS tool ignores these commonly used system calls such as printf, fscanf..
  - Vitis HLS tool defines macro `_SYNTHESIS_`, which allows excluding non-synthesizable code from the design
- Dynamic Memory Usage
  - All the constructs of C are supported in HLS provided they are statically defined at compile time
  - If a function is not fully realized, it cannot be synthesized : `malloc()`, `alloc()`, `free()`
  - HLS tool doesn't support C++ objects that are dynamically created or destroyed
- Pointer Limitations
  - No support for general pointer casting but support for pointer casting between native C/C++ types
  - It supports pointer arrays for synthesis, provided that each pointers to a scalar or an array of scalar. Arrays of pointers cannot point to additional pointers
  - Function points are not supported
- Recursive Functions
- Standard Template Libraries

# Library Support

- ▶ HLS C library allow common hardware design constructs and functions to be easily modeled in C and synthesized to RTL
- ▶ Use each of the C library in the design by including the library header file
- ▶ Four types of libraries as follows:
  - Arbitrary Precision Data Types Library
  - HLS Math library (hls\_math.h)
  - HLS Streaming Library
  - HLS IP libraries
  - Vitis Vision Library



# Data Types and Bit-Accuracy

# Data Types and Bit-Accuracy

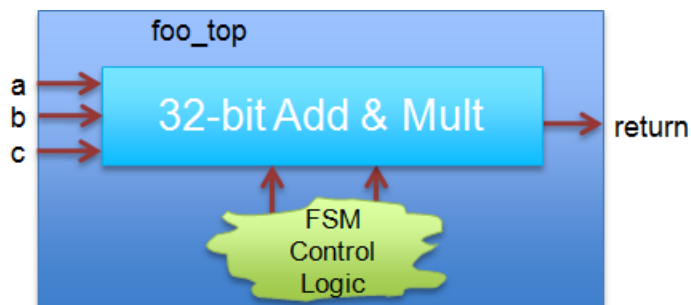
- ▶ C and C++ have standard types created on the 8-bit boundary
  - char (8-bit), short (16-bit), int (32-bit), long long (64-bit)
    - Also provides stdint.h (for C), and stdint.h and cstdint (for C++)
    - Types: int8\_t, uint16\_t, uint32\_t, int\_64\_t etc.
  - They result in hardware which is not bit-accurate and can give sub-standard QoR
- ▶ Vitis HLS tool provides both integer and fixed-point arbitrary precision types for C++
  - Allow any arbitrary bit-width to be specified
  - Designers can:
    - Improve the QoR of the hardware by specifying exact data widths
    - Simulate the design to ensure that there is no loss of accuracy

# Why is arbitrary precision Needed?

## ► Code using native C int type

```
int foo_top(int a, int b, int c)
{
    int sum, mult;
    sum=a+b;
    mult=sum*c;
    return mult;
}
```

Synthesis →

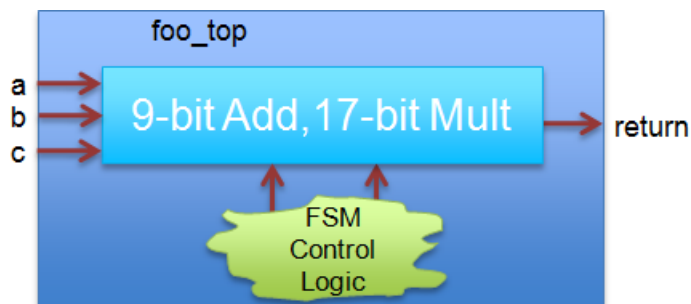


## ► However, if the inputs will only have a max range of 8-bit

- Arbitrary precision data-types should be used

```
int17 foo_top(int8 a, int8 b, int8 c)
{
    int9 sum;
    int17 mult
    sum=a+b;
    mult=sum*c;
    return mult;
}
```

Synthesis →

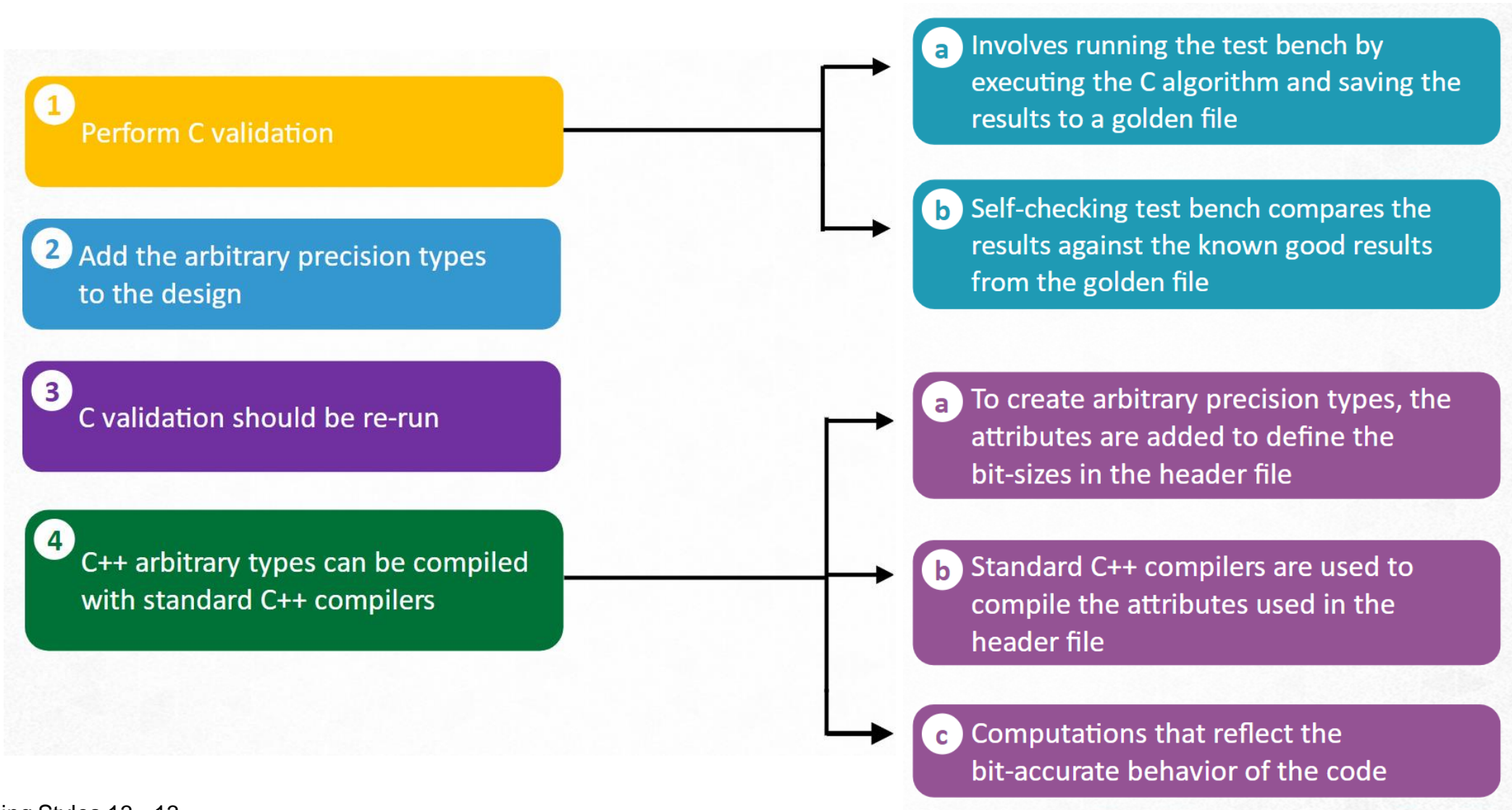


- It will result in smaller & faster hardware with the full required precision
- Full precision can be simulated/validated with C simulation and hardware will behave the same

# HLS & C Types

- ▶ There are 3 basic types you can use for HLS
  - Standard C/C++ Types
  - Vitis HLS enhancements to C++: `ap_int`,
  - Vitis HLS enhancements to C++: `ap_fixed`

# Arbitrary Precision Flow



# Arbitrary Precision : C++ ap\_int types

## ► For C++

- Vitis HLS types ap\_int can be used
- Range: 1 to 1024 bits
  - Signed: ap\_int<W>
  - Unsigned: ap\_uint<W>
- The bit-width is specified by W
- User must include the ap\_int.h header file in the source files

```
#include ap_int.h
```

Include header file

```
void foo_top (...) {
```

```
    ap_int<9>
```

```
    var1;
```

```
    // 9-bit
```

```
    ap_uint<10>
```

```
    var2;
```

```
    // 10-bit unsigned
```

## ► C++ compilation

- Use g++ at the Vitis HLS CLI (shell)
  - Include the path to the Vitis HLS tool header file

```
shell> g++ -o my_test test.c test_tb.c -I$VITIS_HLS_HOME/include
```

# AP\_INT operators & conversions

► Fully Supported for all Arithmetic operator

Operations	
Arithmetic	+ - * / % ++ --
Logical	~ !
Bitwise	&   ^
Relational	> < <= >= == !=
Assignment	*= /= %+= += -= <<= >>= &= ^=  =

► Methods for type conversion

Methods		Example
To integer	Convert to an integer type	res = var.to_int();
To unsigned integer	Convert to an unsigned integer type	res = var.to_uint();
To 64-bit integer	Convert to a 64-bit long long type	res = var.to_int64();
To 64-bit unsigned integer	Convert to an unsigned long long type	res = var.to_uint64();
To double	Convert to double type	res = var.to_double();

# AP\_INT Bit Manipulation methods

Methods		Example
Length	Returns the length of the variable.	res=var.length;
Concatenation	Concatenation low to high	res=var_hi.concat(var_lo); Or res= (var_hi,var_lo)
Range or Bit-select	Return a bit-range from high to low or a specific bit.	res=var.range(high bit,low bit); Or res=var[bit-number]
(n)and_reduce	(N)And reduce all bits.	bool t = var.and_reduce();
(n)or_reduce	(N)Or reduce all bits	bool t = var.or_reduce();
X(n)or_reduce	X(N)or reduce all bits	bool t = var.xor_reduce();
Reverse	Reverse the bits in the variable	var.reverse();
Test bit	Tests if a bit is true	bool t = var.test(bit-number)
Set bit value	Sets the value of a specific bit	var.set_bit(bit-number, value)
Set bit	Set a specific bit to one	var.set(bit-number);
Clear bit	Clear a specific bit to zero	var.clear(bit-number);
Invert Bit	Invert a specific bit	var.invert(bit-number);
Rotate right	Rotate the N-bits to the right	var.rrotate(N);
Rotate left	Rotate the N-bits to the left	var.lrotate(N);
Bitwise Invert	Invert all bits	var.b_not();
Test sign	Test if the sign is negative (return true)	bool t = var.sign();



# Arbitrary Precision : C++ ap\_fixed types

- ▶ Support for fixed point datatypes in C++
  - Include the path to the ap\_fixed.h header file
  - Both signed (ap\_fixed) and unsigned types (ap\_ufixed)

```
#include ap_fixed.h
void foo_top (...) {
    ap_fixed<9, 5, AP_RND_CONV, AP_SAT> var1;           // 9-bit,
                                                         // 5 integer bits, 4 decimal places
    ap_ufixed<10, 7, AP_RND_CONV, AP_SAT> var2;         // 10-bit unsigned
                                                         // 7 integer bits, 3 decimal places
```

- ▶ Advantages of Fixed Point types
  - The result of variables with different sizes is automatically taken care of
  - The binary point is automatically aligned
    - Quantization: Underflow is automatically handled
    - Overflow: Saturation is automatically handled

**Alternatively, make the result variable large enough such that overflow or underflow does not occur**

# AP\_FIXED operators & conversions

- ▶ Fully Supported for all Arithmetic operator

Operations	
Arithmetic	+ - * / % ++ --
Logical	~ !
Bitwise	&   ^
Relational	> < <= >= == !=
Assignment	*= /= %+= += -= <<= >>= &= ^=  =

- ▶ Methods for type conversion

Methods		Example
To integer	Convert to an integer type	res = var.to_int();
To unsigned integer	Convert to an unsigned integer type	res = var.to_uint();
To 64-bit integer	Convert to a 64-bit long long type	res = var.to_int64();
To 64-bit unsigned integer	Convert to an unsigned long long type	res = var.to_uint64();
To double	Convert to double type	res = var.to_double();
To ap_int	Convert to an ap_int	res = var.to_ap_int();

# AP\_FIXED methods

► Methods for bit manipulation

Methods		Example
Length	Returns the length of the variable.	res=var.length;
Concatenation	Concatenation low to high	res=var_hi.concat(var_lo); Or res= (var_hi,var_lo)
Range or Bit-select	Return a bit-range from high to low or a specific bit.	res=var.range(high bit,low bit); Or res=var[bit-number]

# Quantization Modes: Rounding

- ▶ **AP\_RND\_ZERO: rounding to zero**
  - For positive numbers, the redundant bits are truncated
  - For negative numbers, add MSB of removed bits to the remaining bits.
  - The effect is to round towards zero.
    - 01.01 (1.25 using 4 bits) rounds to 01.0 (1 using 3 bits)
    - 10.11 (-1.25 using 4 bits) rounds to 11.0 (-1 using 3 bits)
  
- ▶ **AP\_RND\_CONV: rounded to the nearest value**
  - The rounding depends on the least significant bit
  - If the least significant bit is set, rounding towards plus infinity
  - Otherwise, rounding towards minus infinity
    - 00.11 ( 0.75 using 4-bit) rounds to 01.0 (1.0 using 3-bit)
    - 10.11 (-1.25 using 4-bit) rounds to 11.0 (-1.0 using 3-bit)

# Quantization Modes: Truncation

- ▶ AP\_TRN: truncate
  - Remove redundant bits. Always rounds to minus infinity
  - This is the default.
    - 01.01(1.25) → 01.0 (1)
  
- ▶ AP\_TRN\_ZERO: truncate to zero
  - For positive numbers, the same as AP\_TRN
    - For positive numbers: 01.01(1.25) → 01.0(1)
  - For negative numbers, round to zero
    - For negative numbers: 10.11 (-1.25) → 11.0(-1)

# Overflow Modes

## ▶ Overflow mode

- Determines the behavior when an operation generates more bits than can be satisfied by the MSB

## ▶ Overflow Modes (saturation)

- AP\_SAT, AP\_SAT\_ZERO, AP\_SAT\_SYM

## ▶ Overflow Modes (wrap)

- AP\_WRAP, AP\_WRAP\_SM
- The number of saturation bits, N, is considered when wrapping

# Overflow Mode: Saturation

- ▶ AP\_SAT: saturation
  - This overflow mode will convert the specified value to MAX for an overflow or MIN for an underflow condition
  - MAX and MIN are determined from the number of bits available
- ▶ AP\_SAT\_ZERO: saturates to zero
  - Will set the result to zero, if the result is out of range
- ▶ AP\_SAT\_SYM: symmetrical saturation
  - In 2's complement notation one more negative value than positive value can be represented
  - If it is desirable to have the absolute values of MIN and MAX symmetrical around zero, AP\_SAT\_SYM can be used
  - Positive overflow will generate MAX and negative overflow will generate -MAX
    - 0110(6) => 011(3)
    - 1011(-5) => 101(-3)

# Overflow Mode: Wrap Sign Magnitude

## ▶ AP\_WRAP\_SM, $N = 0$

- This mode uses sign magnitude wrapping
- Sign bit set to the value of the least significant deleted bit
- If the most significant remaining bit is different from the original MSB, all the remaining bits are inverted
- IF MSBs are same, the other bits are copied over
  - Step 1: First delete redundant MSBs. 0100(4) => 100(-4)
  - Step 2: The new sign bit is the least significant bit of the deleted bits. 0 in this case
  - Step 3: Compare the new sign bit with the sign of the new value
- If different, invert all the numbers. They are different in this case
  - 011 (3) 11

## ▶ AP\_WRAP\_SM, $N > 0$

- Uses sign magnitude saturation
- Here N MSBs will be saturated to 1
- Behaves similar to case where  $N = 0$ , except that positive numbers stay positive and negative numbers stay negative



# Floating Point Support

- ▶ Synthesis for floating point
  - Data types (IEEE-754 standard compliant)
    - Single-precision
      - 32 bit: 24-bit fraction, 8-bit exponent
    - Double-precision
      - 64 bit: 53-bit fraction, 11-bit exponent
- ▶ Support for Operators
  - Vitis HLS supports the Floating Point (FP) cores for each Xilinx technology
    - If Xilinx has a FP core, Vitis HLS supports it
    - It will automatically be synthesized
  - If there is no such FP core in the Xilinx technology
    - Design should be re-coded to fixed-point types

# Using Floating-Point Types Example

- ▶ Floats and doubles are commonly used with `math.h` for C and `cmath.h` for C++

<code>double</code>	<code>foo_d = 3.1459;</code>	<div>Using <code>ap_fixed</code> requires:</div> <ul style="list-style-type: none"><li>• C++</li><li>• <code>\$VITIS_HLS_HOME/include/ap_fixed.h</code></li></ul>
<code>float</code>	<code>foo_f = 3.1459 ;</code>	
<code>ap_fixed&lt;14,4&gt;</code>	<code>foo_fx = -1.4142;</code>	
<code>int</code>	<code>foo_i = 42;</code>	

- ▶ While converting C functions to C++ to take advantage of `math.h` support, be sure that the new C++ code compiles correctly before synthesizing with the Vitis HLS tool

<code>extern "C" float sqrtf(float);</code>	<div>Required if it is a C++ function</div>
---	---

- ▶ Using a `sqrtf`(function)
  - Remove the need for the type converters in the hardware/Save the area and improves the timing

<code>double</code>	<code>var_d = sqrt(foo_d);</code>	<code>//64-bit sqrt core</code>	<div>Using <code>sqrt</code> instead of <code>sqrtf</code> would imply a single-to-double conversion and back</div>
<code>float</code>	<code>var_f = sqrtf(foo_f);</code>	<code>//This will lead to a single precision core</code>	
	<code>var_f = sqrt(foo_f);</code>	<code>//Still 64-bit, with format conversion cores(single to double and back)</code>	
<code>ap_fixed&lt;14,4&gt;</code>	<code>var_fx = sqrtf(foo_fx);</code>	<code>//fixed-point to single precision conversion</code>	
		<code>//Fixed -&gt; 32-bt sqrt core -&gt; floating to fixed conversion</code>	
		<code>//int to float conversion</code>	
<code>int</code>	<code>var_i = sqrtf(foo_i);</code>	<code>Int -&gt; 32-bit sqrt -&gt; float to int</code>	

# Support for Math Functions

- ▶ Many other functions from the C/C++ standard math library for which are no floating-point operator cores available
  - These functions are implemented in a bit-approximate manner
  - Results for these functions may differ within a few units of least precision(ULP) to the C/C++ standards
- ▶ Stand C math library
  - C simulation results and the C/RTL co-simulation results may be different
- ▶ Hls\_math.h library
  - C simulation and C/RTL co-simulation results are identical
  - However, there results of C simulation are not the same as those using standard C libraries
- **Recommendation**
  - Replace math.h or cmath.h with hls\_math.h or to add the files hls\_lib.c and keep math/cmath.h

# Support for Math Functions

Function	Float	Double	Accuracy (ULP)	Implementation Style
fabs	Supported	Supported	Exact	Synthesized
fabsf	Supported	Not Applicable	Exact	Synthesized
floorf	Supported	Not Applicable	Exact	Synthesized
fmax	Supported	Supported	Exact	Synthesized
fmin	Supported	Supported	Exact	Synthesized
logf	Supported	Not Applicable	1	Synthesized
floor	Supported	Supported	Exact	Synthesized
fpclassify	Supported	Supported	Exact	Synthesized
isfinite	Supported	Supported	Exact	Synthesized
isinf	Supported	Supported	Exact	Synthesized
isnan	Supported	Supported	Exact	Synthesized
isnormal	Supported	Supported	Exact	Synthesized
log	Supported	Supported	1 for float, 16 for double	Synthesized
log10	Supported	Supported	2 for float, 3 for double	Synthesized
1/x (reciprocal)	Supported	Supported	Exact	LogiCORE IP
recip	Supported	Supported	1	Synthesized
recipf	Supported	Not Applicable	1	Synthesized
round	Supported	Supported	Exact	Synthesized
rsqrt	Supported	Supported	1	Synthesized
rsqrtf	Supported	Not Applicable	1	Synthesized
1/sqrt (reciprocal sqrt)	Supported	Supported	Exact	LogiCORE IP
signbit	Supported	Supported	Exact	Synthesized
sin	Supported	Supported	10	Synthesized
sincos	Supported	Supported	1 for float, 5 for double	Synthesized
sincosf	Supported	Not Applicable	1	Synthesized
sinf	Supported	Not Applicable	1	Synthesized
sinhf	Supported	Not Applicable	6	Synthesized
sqrt	Supported	Supported	Exact	LogiCORE IP
tan	Supported	Supported	20	Synthesized
tanf	Supported	Not Applicable	3	Synthesized
trunc	Supported	Supported	Exact	Synthesized



# Using Pointers with Limitation

# Using Pointers

Used extensively in C code and are well supported for synthesis

Used as arguments to the top-level function

## Important to understand

How pointers are implemented during synthesis because they can sometimes cause issues in achieving the desired RTL interface

If the designer uses a pointer in multiple functions, then Vitis HLS inlines all these functions and tries to increase the run time

# Using Pointers

Structs as Pointers	Pointer Arithmetic	Converting Pointers Using Malloc
<p>Structs can be implemented differently as a pointer or as a pass-by-value argument</p>	<p>Introducing the pointer arithmetic limits the possible interfaces that can be synthesized in RTL</p> <p>No sequential access to pointer data</p> <p>This interface supplies an address with which to index the data when the data is accessed</p>	<p>Fixed-sized resources can be created, and the existing pointer can simply be made to point to these fixed-sized resources</p> <p>This technique can prevent manual recoding of the existing design</p>
Multi-Access Pointers		
<p>Multiple accesses occur when a pointer is read from or written to multiple times in the same function</p> <p>Volatile qualifier can be used on any function argument that is accessed multiple times</p> <p>In the case of multi-access pointers, C code cannot be rigorously validated with C simulation</p> <p>Requires a depth specification for RTL simulation</p>		

# Structs as Pointers Passed as a Pointer



8-bit port for each variable with 'char' data type

Port size for the B variable is  $4 * 8 = 32$  bits

When a struct is passed as a pointer, the data will be accessed in a **pass-by-reference method** → port created for all members of the struct is 8 bits each



# Structs as Pointers Passed by Pass-by-Value Method



When a struct is passed by pass-by-value method, the data will be accessed in a **pass-by-value method** → port created for A and C = 8 bits, but the port created for B = 32 bits

Since B is an array of 4 char members, each member of the array takes 8 bits, and B takes a total of 32 bits

# Structs as Pointers

No limitations in the size or complexity of the structs

There can be as many array dimensions and as many members in a struct as required

Only limitation with the implementation of the structs is when the arrays are to be implemented as streaming

# Pointer Arithmetic

```
#include "bus.h"

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

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

Plus  
Testbench

```
# int main () {
    int d[5];
    int i;

    for (i=0; i<5; i++){
        d[i] = i+5;
    }

    foo(d);

    // Check results
    ...
}
```

Simple pointer arithmetic used to accumulate the data values

Introducing the pointer arithmetic limits the possible interfaces that can be synthesized in RTL

Wire, handshake, and FIFO interfaces → data must arrive and is written in sequence, starting from element zero

# Pointer Arithmetic

```
#include "bus.h"

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

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



```
# int main () {
    int d[5];
    int i;

    for (i=0; i<5; i++){
        d[i] = i+5;
    }

    foo(d);

    // Check results
    ...
}
```



Expected	Actual
-----	-----
res[0]: 6 == d[0]: 6	
res[1]: 13 == d[1]: 13	
res[2]: 21 == d[2]: 21	
res[3]: 30 == d[3]: 30	
res[4]: 9 == d[4]: 9	

Output address 4 is never written to

In the code example → output address 4 is never written

# Pointer Arithmetic

If wire, handshake, and FIFO interfaces are used in this case → RTL co-simulation fails

```
Expected      Actual
.....
res[0]: 6 != d[0]: 5
res[1]:13 != d[1]: 6
res[2]:21 != d[2]: 7
res[3]:30 != d[3]: 8
res[4]: 9 == d[4]: 9
SystemC: simulation stopped by user.
@E [SIM-3] Simulation failed: testbench return error code "1".
@E [SIM-1] *** AutoSim finished: FAIL ***
```

**Only a bus interface can be used  
when I/O pointers use non-trivial  
arithmetic**

- Modify the code with an array on the interface
- Implement this in synthesis with ap\_memory
- Interface can index the data with an address and can perform out-of-order, or nonsequential, accesses

# Converting Pointers Using Malloc

Any system calls that manage the memory allocation are using resources that exist in the memory

System calls are created and released during run time

## To synthesize hardware

Design must be fully self-contained, specifying all the required resources

## To synthesize hardware which uses malloc()

Must be transformed into the equivalent bounded representations of a defined size, such as an array

Coding changes can impact the functionality of the design

# Converting Pointers Using Malloc

## Workaround

1. Add the user-defined macro NO\_SYNTN to the code
  - Used to select between the synthesizable and non-synthesizable versions
2. Enable macro NO\_SYNTN, execute the C simulation, and save the results
3. Disable the macro and execute the C simulation to verify that the results are identical
4. Perform synthesis with the user-defined macro disabled

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

dout_t malloc_removed(din_t din[N], dsel_t width) {
  #ifdef NO_SYNTN
    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;
}
```

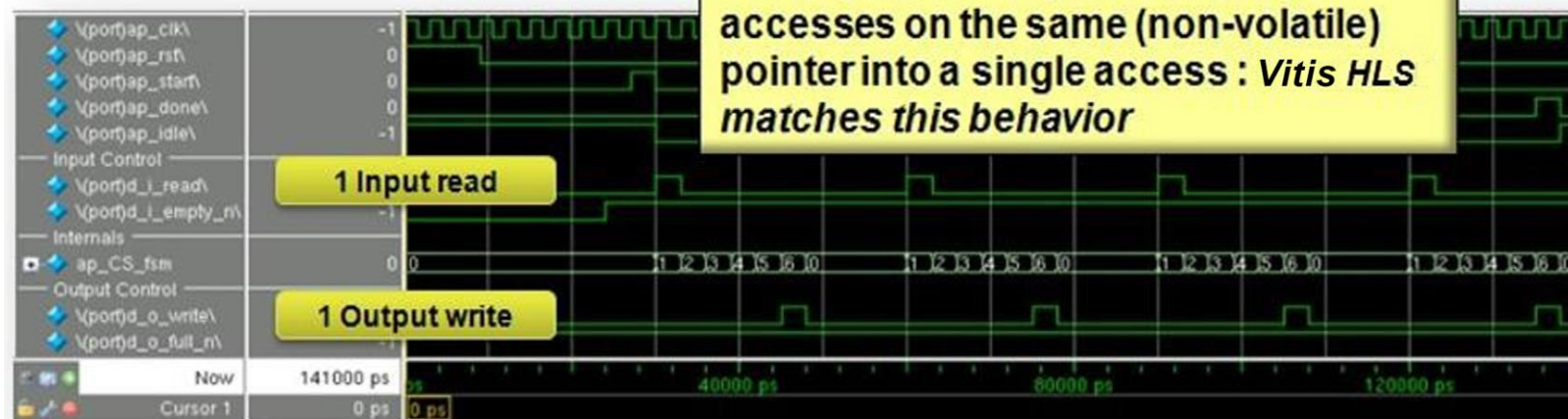


# Multi-Access Pointers

```
void fifo (
    int*d_o,
    int*d_i
){
    static int acc = 0;
    int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

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



- Vitis HLS tool matches the behavior of the gcc compiler
- Optimizes these reads and writes into a single read and a single write operation on each port when the RTL is examined
- To make this design read and write to the RTL ports multiple times, there is a need to use a volatile qualifier

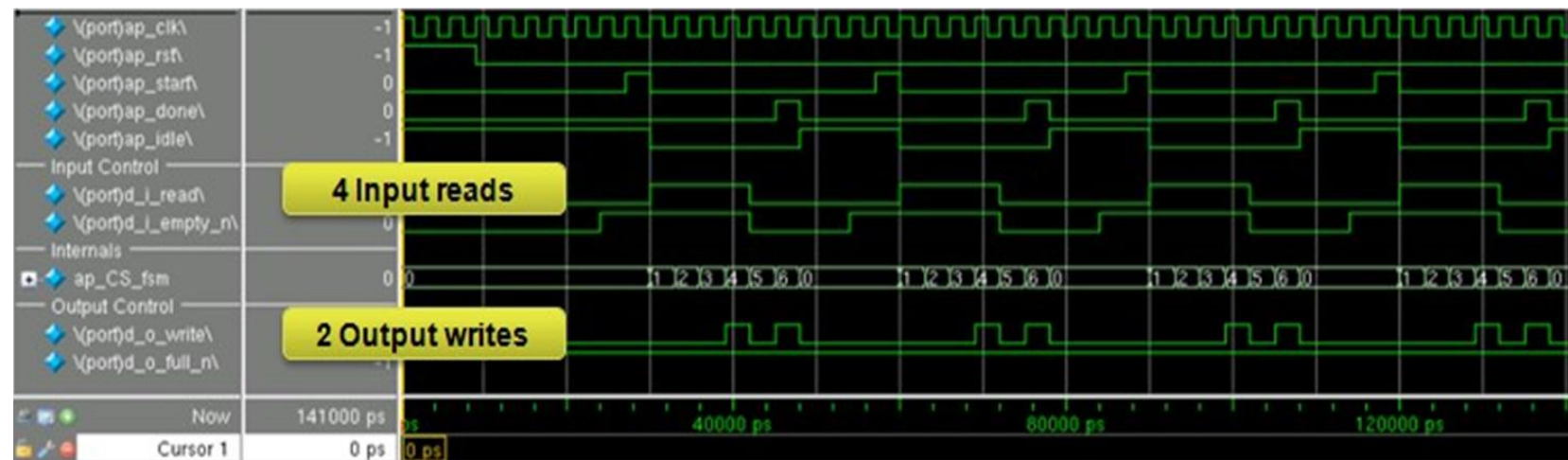


# Multi-Access Pointers

```
void fifo (
    volatile int *d_o,
    volatile int *d_i
){
    static int acc = 0;
    int cnt;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;

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



- Performs four reads, but the same data is read four times
- Performs two separate writes, each with the correct data, but the test bench captures data only for the final write

# Multi-Access Pointers

**SOLUTION**

Additional code can be added to the design under test (DUT)

**Use `__SYNTHESIS__`**

Adds un-synthesizable code to the main code, which will be used just for verification, and it will not be present in the RTL

Not an ideal coding style for verification

**Use `HLS::STREAMS`**

Modifies the main code for using a streaming data type

Streaming data type allows the hardware data to be accurately modeled

# Multi-Access Pointers: Test Bench Depth

```
#include "bus.h"

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

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

**C testbench may correctly  
provide four values**

```
int main () {
    int d[5];
    int i;

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

    foo(d);

    return 0;
}
```

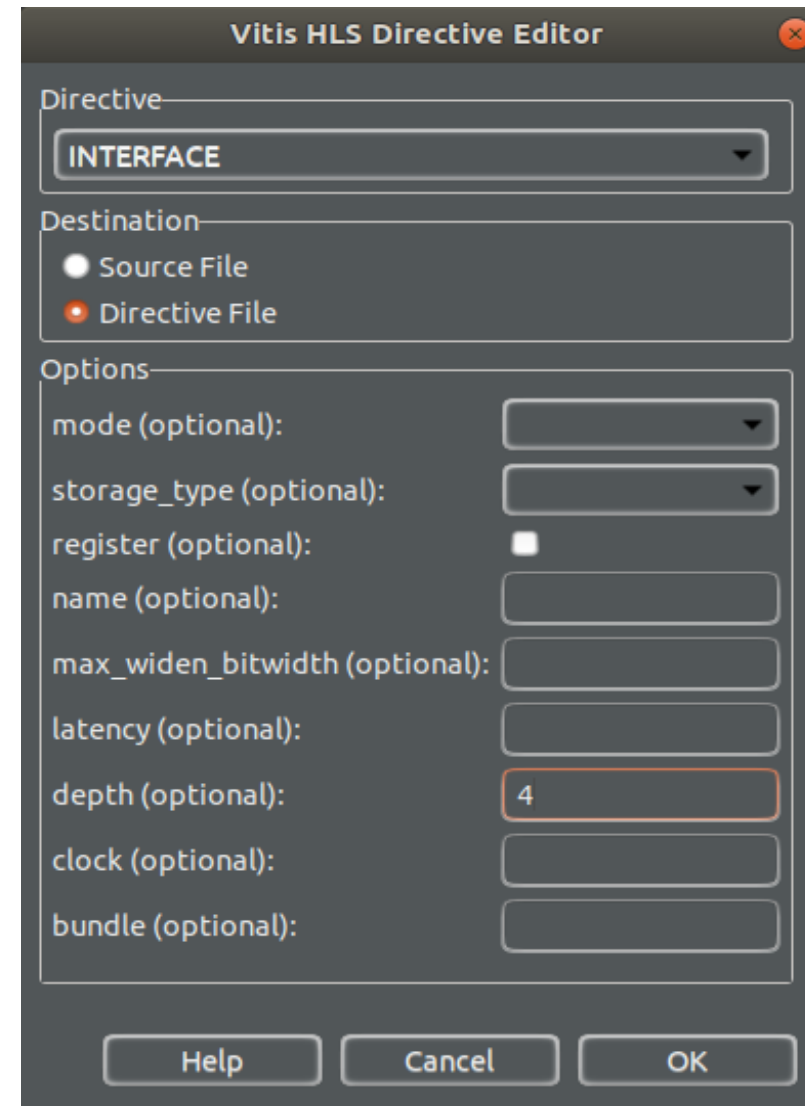
HLS tool does not know how many samples are required unless the interface informs it

Until then, Vitis HLS tool assumes single access has been requested and creates C/RTL co-simulation for only a single input and a single output

When multi-access pointers are used at the interface, Vitis HLS tool must be informed of the maximum number of reads or writes on the interface

# Multi-Access Pointers: Test Bench Depth

- This depth option for the INTERFACE directive can be used to set different values
- Incorrect depth value may result in simulation mismatch
- Vitis HLS tool issues a warning



The screenshot shows the 'Vitis HLS Directive Editor' dialog box. It has a title bar with a close button. The dialog is divided into three main sections: 'Directive', 'Destination', and 'Options'. The 'Directive' section has a dropdown menu currently set to 'INTERFACE'. The 'Destination' section has two radio buttons: 'Source File' (unselected) and 'Directive File' (selected). The 'Options' section contains several labeled input fields: 'mode (optional):' (dropdown), 'storage\_type (optional):' (dropdown), 'register (optional):' (checkbox, unselected), 'name (optional):' (text box), 'max\_widen\_bitwidth (optional):' (text box), 'latency (optional):' (text box), 'depth (optional):' (text box with the value '4' entered and highlighted by a red border), 'clock (optional):' (text box), and 'bundle (optional):' (text box). At the bottom of the dialog are three buttons: 'Help', 'Cancel', and 'OK'.

# Streams

# Designing with Streams

- ▶ Streams can be used instead of multi-access pointers
  - None of the issues
- ▶ Streams simulate like an infinite FIFO in software
  - Implemented as a FIFO of user-defined size in hardware
- ▶ Streams have support for multi-access
  - Streams interface to the testbench
  - Streams can be read in the testbench to check the intermediate values
  - Streams store values: no chance of the volatile effect
- ▶ Streams are supported on the interface and internally
  - Define the stream as static to make it internal only

# Using Streams

- ▶ Streams are C++ classes
  - Modeled in C++ as an infinite depth FIFO
  - Can be written to or read from
- ▶ Ideal for Hardware Modeling
  - Ideal for modeling designs in which the data is known to be streaming (data is always in sequential order)
    - Video or Communication designs typically stream data
  - No need to model the design in C++ as a “frame”
  - Streams allow it to be modeled as per-sample processing
    - Just fill the stream in the test bench
    - Read and write to the stream as if it's an infinite FIFO
    - Read from the stream in the test bench to confirm results
- ▶ Streams are by default implemented as a FIFO of depth 2
  - Can be specified by the user: needed for decimation/interpolation designs

# Stream Example

## ► Create using hls::stream or define the hls namespace

```
#include <ap_int.h>
#include <hls_stream.h>

typedef ap_uint<128> uint128_t;      // 128-bit user defined type

hls::stream<uint128_t> my_wide_stream; // A stream declaration
```

```
#include <ap_int.h>
#include <hls_stream.h>
using namespace hls;                // Use hls namespace

typedef ap_uint<128> uint128_t;      // 128-bit user defined type

stream<uint128_t> my_wide_stream;    // hls:: no longer required
```

## ► Blocking and Non-Block accesses supported

```
// Blocking Write
hls::stream<int> my_stream;

int src_var = 42;
my_stream.write(src_var);
// OR use: my_stream << src_var;
```

```
// Blocking Read
hls::stream<int> my_stream;

int dst_var;
my_stream.read(dst_var);
// OR use: dst_var = my_stream.read();
```

```
hls::stream<int> my_stream;

int src_var = 42;
bool stream_full;

// Non-Blocking Write
if (my_stream.write_nb(src_var)) {
    // Perform standard operations
} else {
    // Write did not happen
}

// Full test
stream_full = my_stream.full();
```

```
hls::stream<int> my_stream;

int dst_var;
bool stream_empty;

// Non-Blocking Read
if (my_stream.read_nb(dst_var)) {
    // Perform standard operations
} else {
    // Read did not happen
}

// Empty test
fifo_empty = my_stream.empty();
```

Stream arrays and structs are not supported for RTL simulation at this time: must be verified manually.

e.g. `hls::stream<uint8_t> chan[4]`



# Summary

# Summary

- ▶ C and C++ have standard types created on the 8-bit boundary
  - char (8-bit), short (16-bit), int (32-bit), long long (64-bit)
- ▶ Arbitrary precision in C++ is supported using `ap_int`
  - Arbitrary precision types can define bit-accurate operators leading to better QoR
- ▶ Fixed point precision is supported in C++
  - Both signed (`ap_fixed`) and unsigned types (`ap_ufixed`)
- ▶ Pointers that are accessed multiple times can introduce unexpected behavior after synthesis
- ▶ Although multi-access pointers are supported, it is highly recommended to implement the behavior by using the `hls::stream` class
- ▶ `malloc` is not supported for synthesis
  - Must be converted to a resource of a defined size → `arrayTrace`



# Thank You

# Disclaimer and Attribution

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

© Copyright 2021 Advanced Micro Devices, Inc. All rights reserved. Xilinx, the Xilinx logo, AMD, the AMD Arrow logo, Alveo, Artix, Kintex, Kria, Spartan, Versal, Vitis, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

