Data Types

Vivado HLS 2013.3 Version

Objectives

- **After completing this module, you will be able to:**
  - State various data types of C, C++, and SystemC are supported
  - Identify advantages and pitfalls of using arbitrary precision
  - List various supported quantization and overflow modes
  - Describe the floating point support

Outline

- *C and C++ Data Types*
- Arbitrary Precision Data Types
- System C Data Types
- Floating Point Support
- Summary

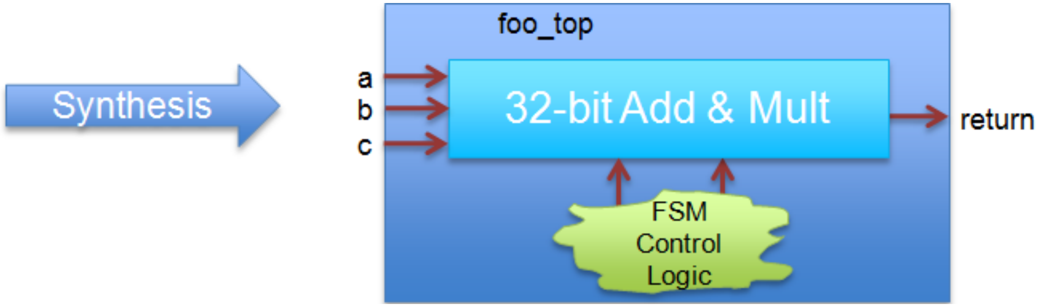
Data Types and Bit-Accuracy

- **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
- **Vivado HLS provides bit-accurate types in both C and C++**
  - Allow any arbitrary bit-width to be specified
  - Hence designers can improve the QoR of the hardware by specifying exact data widths
    - Can be specified in the code and simulated to ensure there is no loss of accuracy

Why is arbitrary precision Needed?

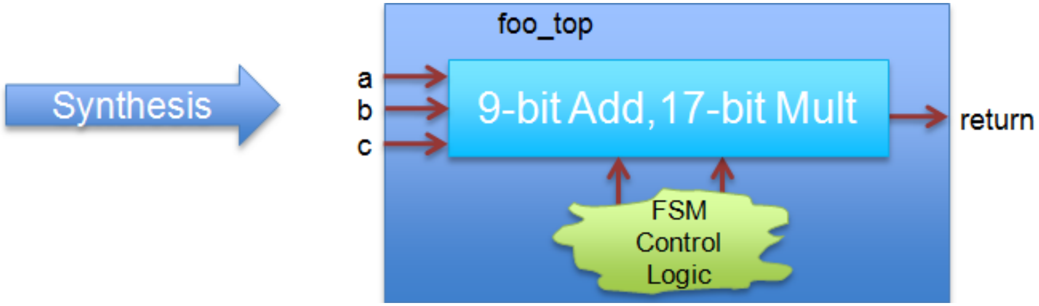
- **Code using native C int type**

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



- **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;
}
```



  – It will result in smaller & faster hardware with the full required precision
  – With arbitrary precision types on function interfaces, Vivado HLS can propagate the correct bit-widths throughout the design

HLS & C Types

- **There are 4 basic types you can use for HLS**
  - Standard C/C++ Types
  - Vivado HLS enhancements to C: apint
  - Vivado HLS enhancements to C++: ap_int, ap_fixed
  - SystemC types

Type of C	C(C99) / C++	Vivado HLS ap_cint (bit-accurate with C)	Vivado HLS ap_int (bit-accurate with C++)	OSCI SystemC (IEEE 1666-2005 :bit-accurate)
Description		Used with standard C	Used with standard C++	IEEE standard
Requires		#include "ap_cint.h"	#include "ap_int.h" #include "ap_fixed.h" #include "hls_stream.h"	#include "systemc.h"
Pre-Synthesis Validation	gcc/g++		g++ Vivado HLS GUI	g++ Vivado HLS GUI
Fixed Point	NA	NA	ap_fixed	#define SC_INCLUDE_FX sc_fixed
Signal Modeling	Variables	Variables	Variables Streams	Signals, Channels, TLM (1.0)

Outline

- C and C++ Data Types
- ***Arbitrary Precision Data Types***
- System C Data Types
- Floating Point Support
- Summary

Arbitrary Precision : C apint types

- **For C**
  - Vivado HLS types apint can be used
  - Range: 1 to 1024 bits
  - Specify the integers as shown and just use them like any other variable

```
#include ap_cint.h

void foo_top (…) {

   int9          var1;                    // 9-bit
   uint10        var2;                    // 10-bit unsigned
```

Include header file

- **There are two issues to be aware of**
  - C compilation : YOU MUST use apcc to simulate (no debugger support)
  - Be aware of integer promotion issues

Failure to use apcc to compile the C will result in INCORRECT results

This only applies to C NOT C++ or SystemC

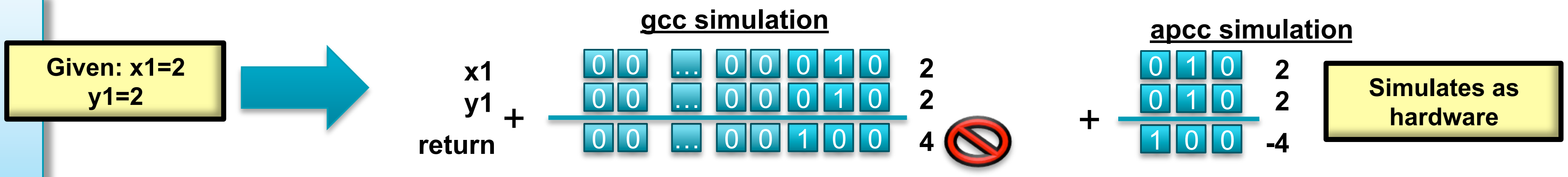
Using apcc

- **apcc**
  - Command line compatible with gcc
  - Required to support arbitrary precision for C
  - Use apcc at the Vivado HLS CLI (shell)

```
shell> apcc –o my_test test.c test_tb.c
```

- **apcc understands bit-accurate types**

```
#include "ap_cint.h"
int3 ex_bit_accurate (
  int3 x1,
  int3 y1
  ) {
  return x1+y1;
}
```

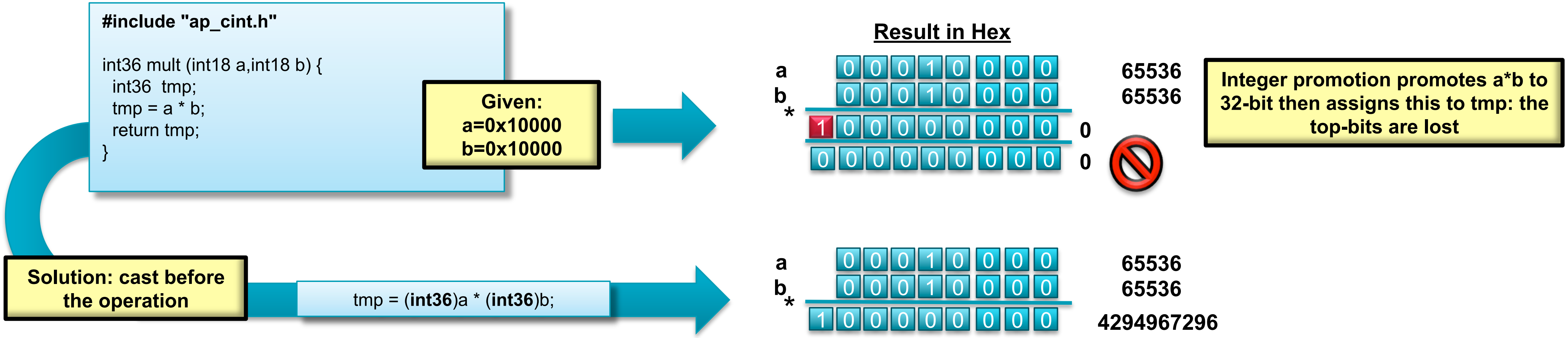


  - Once you create bit-accurate types you must re-validate the C
  - It's the only way to discover rounding and truncation issues
    - It's fast in C !!!

XILINX ALL PROGRAMMABLE.

Integer Promotion

- **Integer promotion**
  - The apcc utility must still obey standard C/gcc rules and protocols
  - Integer promotion:
    - If the operator result is a larger type ➔
    - The result is promoted to the target type (on 8, 16, 32 or 64 boundaries)



XILINX ALL PROGRAMMABLE.

C apint types: Bit-Selection & Manipulation

Function		Example
Length	Returns the length of the variable.	res=apint_bitwidthof(var);
Concatenation	Concatenation low to high	res=apint_concatenate(var_high, var_low)
Get a range	Return a bit-range from high to low.	res= apint_get_range(var, high,low)
Set a range	Reserve the bits in the variable	apint_set_range(res, high, low, res)
(n)and_reduce	(N)And reduce all bits.	bool t = apint_(n)and_reduce(var);
(n)or_reduce	(N)Or reduce all bits	bool t = apint_(n)or_reduce(var);
X(n)or_reduce	X(N)or reduce all bits	bool t = apint_x(n)or_reduce(var);
Get a bit	Get a specific bit	res=apint_get_bit(var, bit-number)
Set bit value	Sets the value of a specific bit	apint_set_bit(res, bit-number)
Print value	Print the value of an apint variable	apint_print(int#N value, int radix));
Print value to file	Print the value of an apint variable to a file	apint_fprint(FILE* file, int#N value, int radix)

Arbitrary Precision : C++ ap_int types

- **For C++**
  - Vivado 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

Include header file

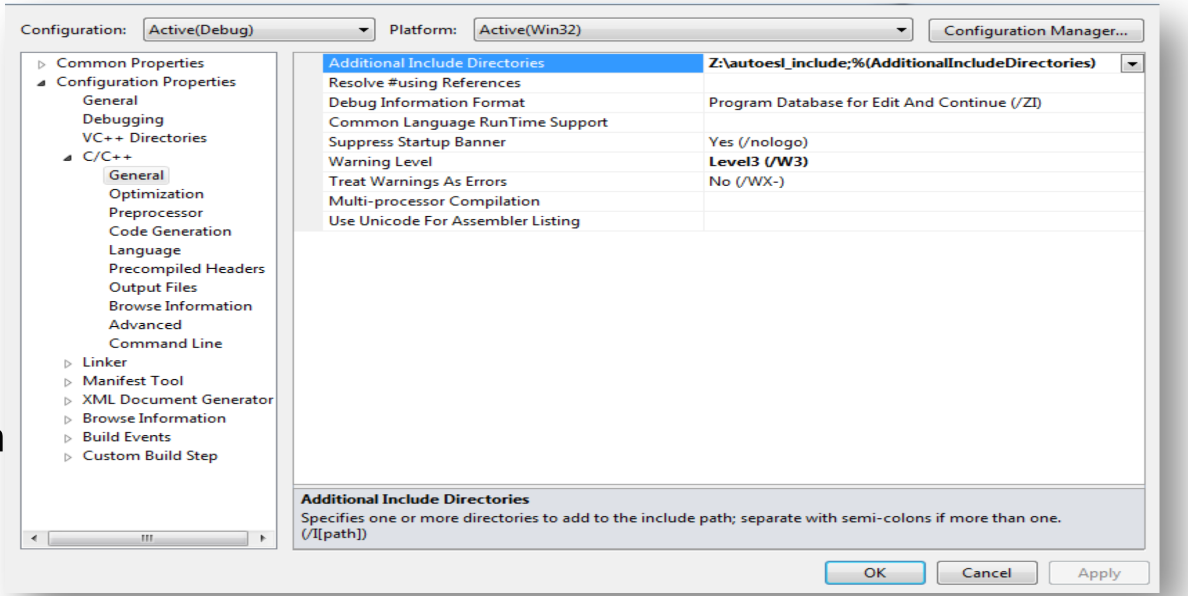
```
#include ap_int.h

void foo_top (…) {

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

- **C++ compilation**
  - Use g++ at the Vivado HLS CLI (shell)
    - Include the path to the Vivado HLS header file

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

Microsoft Visual Studio Support

- **C++ Arbitrary Precision Types are supported in Microsoft Visual Studio Compiler**
  - Simply include the Vivado HLS directory $(VIVADO_HLS_HOME)/include
  - **Note**: C designs using arbitrary precision types (apint) must still use apcc
- **C++ Designs using AP_INT types**
  - In the MVS Project
    - Click Project
    - Click Properties
    - In the panel that shows up, select C/C++
    - Select general
    - Click on additional include directories and add the path



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 a 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.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	Reserve 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)

$VIVADO_HLS_HOME/include/ap_fixed.h

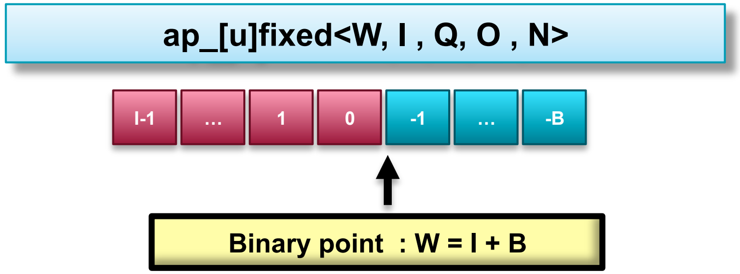
```
#include ap_fixed.h

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

Definition of ap_fixed type

- **Fixed point types are specified by**
  - Total bit width (W)
  - The number of integer bits (I)
  - The quantization/rounding mode (Q)
  - The overflow/saturation mode (O)



	Description	
W	Word length in bits	
I	The number of bits used to represent the integer value (the number of bits above the decimal point)	
Q	Quantization mode (modes detailed below) dictates the behavior when greater precision is generated than can be defined by the LSBs.	
	AP_Fixed Mode	**Description**
	AP_RND	Rounding to plus infinity
	AP_RND_ZERO	Rounding to zero
	AP_RND_MIN_INF	Rounding to minus infinity
	AP_RND_INF	Rounding to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity
	AP_TRN_ZERO	Truncation to zero (default)
O	Overflow mode (modes detailed below) dictates the behavior when more bits are required than the word contains.	
	AP_Fixed Mode	**Description**
	AP_SAT	Saturation
	AP_SAT_ZERO	Saturation to zero
	AP_SAT_SYM	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	The number of saturation bits in wrap modes.	

Quantization Modes

- **Quantization mode**
  - Determines the behavior when an operation generates more precision in the LSBs than is available
- **Quantization Modes (rounding):**
  - AP_RND, AP_RND_MIN_IF, AP_RND_IF
  - AP_RND_ZERO, AP_RND_CONV
- **Quantization Modes (truncation):**
  - AP_TRN, AP_TRN_ZERO

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

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 a 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.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]

Fixed Point Math Functions (Starting 2013.3)

- **The hls_math.h library**
  - Now includes fixed-point functions for sin, cos and sqrt

Function	*Type*	Accuracy (ULP)	Implementation Style
cos	ap_fixed<32,I>	16	Synthesized
sin	ap_fixed<32,I>	16	Synthesized
sqrt	ap_fixed<W,I> ap_ufixed<W,I>	1	Synthesized

  - The sin and cos functions are all 32-bit ap_fixed<32,Int_Bit>
    - Where Int_Bit specifies the number of integer bits
  - The sqrt function is any width but must have a decimal point
    - Cannot be all intergers or all bits
  - The accuracy above is quoted with respect to the equivalent floating point version

Outline

- C and C++ Data Types
- Arbitrary Precision Data Types
- *System C Data Types*
- Floating Point Support
- Summary

Arbitrary Precision : SystemC

- **SystemC is an IEEE standard (IEEE 1666)**
  - C++ class libraries
  - Allows design and simulation with concurrency
  - Provides a library of arbitrary precision types
    - sc_int, sc_uint, sc_bigint (int > 64 bit), sc_fixed, etc.
- **SystemC support**
  - Vivado HLS supports SystemC 1.3 Synthesizable subset[1]
- **SystemC Compilation**
  - Compile with g++
  - Include the SystemC files from the Vivado HLS tree

```
shell> g++ –o my_test test.c test_tb.c \
                                -I$Vivado HLS_HOME\Win_x86\tools\systemc\include \
                                -lsystemc  \
                                -L$Vivado HLS_HOME\Win_x86\tools\systemc\include\lib
```

- **SC Types**
  - Can be used in C++ designs without the need to convert the entire design to SystemC

Outline

- C and C++ Data Types
- Arbitrary Precision Data Types
- System C Data Types
- ***Floating Point Support***
- Summary

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**
  - Vivado HLS supports the Floating Point (FP) cores for each Xilinx technology
    - If Xilinx has a FP core, Vivado HLS supports it
    - It will automatically be synthesized
  - If there is no such FP core in the Xilinx technology, it will not be in the library
    - The design will be still synthesized

Floating Point Cores

Core	7 Series	Virtex-6	Virtex-5	Virtex-4	Spartan-6	Spartan-3
FAddSub	X	X	X	X	X	X
FAddSub_nodsp	X	X	X	-	-	-
FAddSub_fulldsp	X	X	X	-	-	-
FCmp	X	X	X	X	X	X
FDiv	X	X	X	X	X	X
FMul	X	X	X	X	X	X
FMul_nodsp	X	X	X	-	X	X
FMul_meddsp	X	X	X	-	X	X
FMul_fulldsp	X	X	X	-	X	X
FMul_maxdsp	X	X	X	-	X	X
FRSqrt	X	X	X	-	-	-
FRSqrt_nodsp	X	X	X	-	-	-
FRSqrt_fulldsp	X	X	X	-	-	-
FRecip	X	X	X	-	-	-
FRecip_nodsp	X	X	X	-	-	-
FRecip_fulldsp	X	X	X	-	-	-
FSqrt	X	X	X	X	X	X
DAddSub	X	X	X	X	X	X
DAddSub_nodsp	X	X	X	-	-	-
DAddSub_fulldsp	X	X	X	-	-	-
DCmp	X	X	X	X	X	X
DDiv	X	X	X	X	X	X
DMul	X	X	X	X	X	X
DMul_nodsp	X	X	X	-	X	X
DMul_meddsp	X	X	X	-	-	-
DMul_fulldsp	X	X	X	-	X	X
DMul_maxdsp	X	X	X	-	X	X
DRSqrt	X	X	X	X	X	X
DRecip	X	X	X	-	-	-
DSqrt	X	X	X	-	-	-

Support for Math Functions

More Details are available in the Coding Style Guide chapter in the User Guide

- **Vivado HLS provides support for many math functions**
  - Even if no floating-point core exists
  - These functions are implemented in a bit-approximate manner
  - The results may differ within a few Units of Least Precision (ULP) to the C/C++ standards
- **Use math.h (C) or cmath.h (C++)**
  - The functions will be synthesized automatically
  - The C simulation results may differ from the RTL simulation results
  - Use a test bench which checks for ranges: not == or !=
- **Replace math.h or cmath.h with Vivado HLS header file “hls_math.h” Or keep math/ cmath and “add_files hls_lib.c”**
  - The C simulation will match the RTL simulation
  - The C simulation may differ from the C simulation using math/cmath (or math/cmath without hls_lib.c)

Supported Math Functions

- **Floating C point functions ***f**
  - There is no double-precision implementation
  - C++ functions will overload as per the C++ standard
    - Can be used with double or single precision
- **More specific details are in the User Guide**
  - Refer to the Coding Style Guide chapter: C Libraries

Function	Float	Double	Accuracy (ULP)	LogicCore
ceilf	Supported	Not Applicable	Exact	Not Supported
copysignf	Supported	Not Applicable	Exact	Not Supported
fabsf	Supported	Not Applicable	Exact	Not Supported
floorf	Supported	Not Applicable	Exact	Not Supported
logf	Supported	Not Applicable	1 to 5	Not Supported
cosf	Supported	Not Applicable	1 to 100	Not Supported
sinf	Supported	Not Applicable	1 to 100	Not Supported
abs	Supported	Supported	Exact	Not Supported
ceil	Supported	Supported	Exact	Not Supported
copysign	Supported	Supported	Exact	Not Supported
cos	Supported	Supported	2 for float. 5 for double	Not Supported
fabs	Supported	Supported	Exact	Not Supported
floor	Supported	Supported	Exact	Not Supported
fpclassify	Supported	Supported	Exact	Not Supported
isfinite	Supported	Supported	Exact	Not Supported
isinf	Supported	Supported	Exact	Not Supported
isnan	Supported	Supported	Exact	Not Supported
isnormal	Supported	Supported	Exact	Not Supported
log	Supported	Supported	1 for float, 16 for double	Not Supported
log10	Supported	Supported	1 for float, 16 for double	Not Supported
recip	Supported	Supported	Exact	Supported
round	Supported	Supported	Exact	Not Supported
rsqrt	Supported	Supported	Exact	Supported
signbit	Supported	Supported	Exact	Not Supported
sin	Supported	Supported	2 for float, 5 for double	Not Supported
sqrt	Supported	Supported	Exact	Supported
trunc	Supported	Supported	Exact	Not Supported

For more information on floating point refer to Application Note **Floating Point Design with Vivado HLS**

Example on using Floating Point Types

- **The following highlights some typical use scenarios**
  - Example values

```
double              foo_d   = 3.1459;
float               foo_f   = 3.1459;
ap_fixed<14,4>      foo_fx  =  -1.4142;
int                 foo_i   = 42;
```

> Using ap_fixed requires:
> • C++
> • $Vivado HLS_HOME/include/ap_fixed.h

- **When using sqrt() function**
  - It is from math.h which is a C function, not C++

```
extern "C" float sqrtf(float);
```

> Required if it's a C++ function

- **Understand that sqrt() is 64-bit and sqrtf() is 32-bit**

```
double    var_d = sqrt(foo_d);     // 64-bit sqrt core
float     var_f = sqrtf(foo_f);    // This will lead to a single precision sqrt core

          var_f = sqrt(foo_f);     // Still 64-bit, with format conversion cores (single to double and back)
```

- **Type conversions can be used**

```
ap_fixed<14,4>      var_fx = sqrtf(foo_fx);    // fixed-point to single precision conversion
                                               // Fixed → 32-bit sqrt core → float to fixed conversion
int                 var_i = sqrtf(foo_i);      // int to float conversion
                                               // Int → 32-bit sqrt → float to int
```

> Using sqrt instead of sqrtf would imply a single to double conversion and back

Outline

- C and C++ Data Types
- Arbitrary Precision Data Types
- System C Data Types
- Floating Point Support
- ***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)
- **Vivado HLS supports SystemC 1.3 Synthesizable subset**
- **Arbitrary precision in C is supported using apint and ap_int in C++**
  - Compile using apcc for arbitrary precision
  - 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)

Summary

- **Various quantization and overflow modes supported**
  - Quantization
    - AP_RND, AP_RND_ZERO, AP_RND_MIN_INF, AP_RND_INF, AP_RND_CONV, AP_TRN, AP_TRN_ZERO
  - Overflow
    - AP_SAT, AP_SAT_ZERO, AP_SAT_SYM, AP_WRAP, AP_WRAP_SYM
- **Both single- and double-precision floating point data types are supported**
  - If a corresponding floating point core is available then it will automatically be used
  - If floating point core is not available then Vivado HLS will generate the RTL model