Xilinx HLS

- High Level Synthesis -

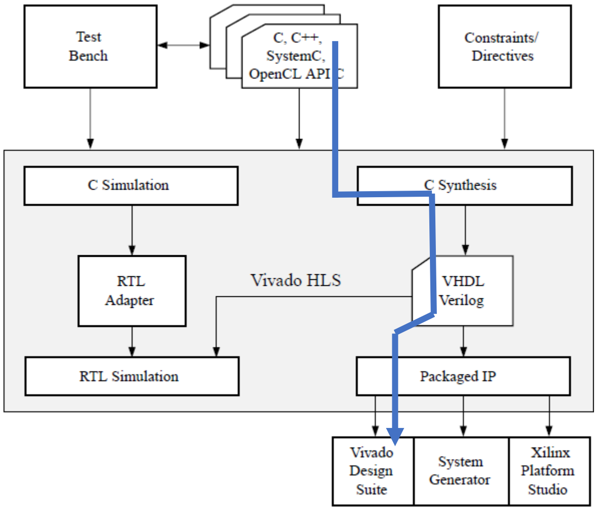
2020

Ando Ki, Ph.D.
adki@future-ds.com

Table of contents

- Vivado HLS
  - Design flow
  - Synthesis basic
  - Vivado HLS pragmas by type
- Optimization basics
  - logic expression
  - pipelining
  - unrolling
  - dataflow
  - array reshaping
- Interfaces
- High-level synthesis C libraries
- High-level synthesis coding styles

Vivado HLS design flow



ANSI-C (GCC 4.6)
C++ (G++ 4.6)
SystemC (IEEE 1666-2006, version 2.2)

C libraries to extend the standard C languages:
- Arbitrary precision data types
- Half-precision (16-bit) floating-point data types in addition to single and double
- Math operations
- Xilinx IP functions, (e.g., FFT, FIR)

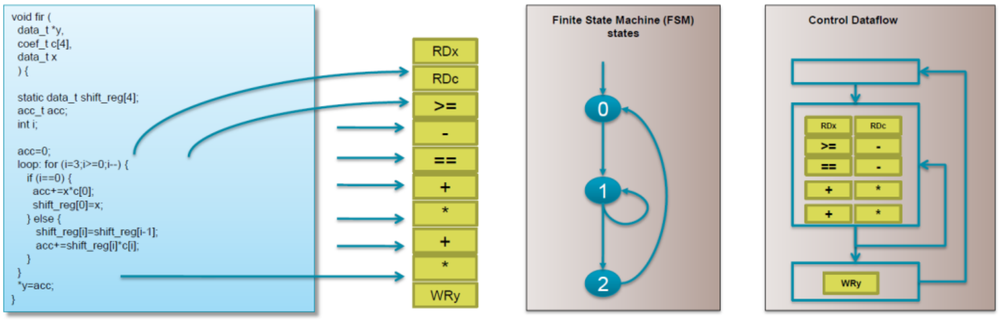
VHDL (IEEE 1076-2000)
Verilog (IEEE 1364-2001)

Not support followings
- Dynamic memory allocation
- OS operations

What HLS does

- Operator extraction, control & datapath synthesis

Code **Operations** **Control Behavior** **Control & Datapath Behavior**



```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i]*c[i];
    }
  }
  *y=acc;
}
```

From any C code example ..

Operations are extracted…

The control is known

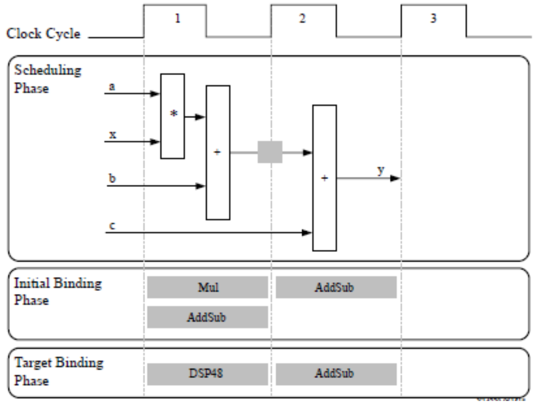
A unified control dataflow behavior is created.

High-Level Synthesis Basics (1/2)

- HLS phases
  - Scheduling
    - Determines operations (i.e., operators)
      - For faster device or sufficient clock period, more operations within a cycle
      - For insufficient clock period (i.e., shorter or a slow FPGA), schedule operations over more than one cycle.
  - Binding
    - Determines hardware resource to use
  - Control logic extraction
    - Create a FSM (Finite State Machine)

```
int foo(char x, char a, char b, char c) {
    return x*a+b+c;
}
```

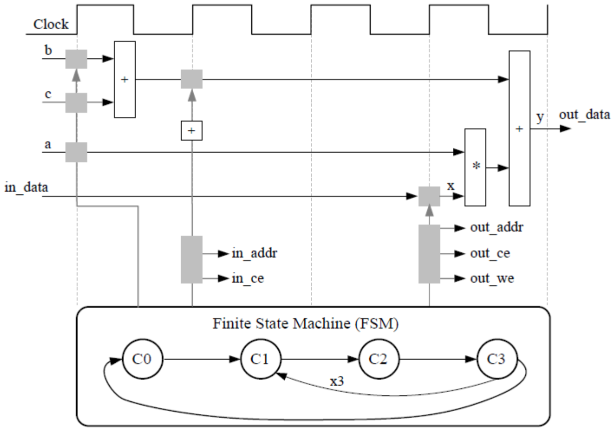
No scheduler is required for this function.



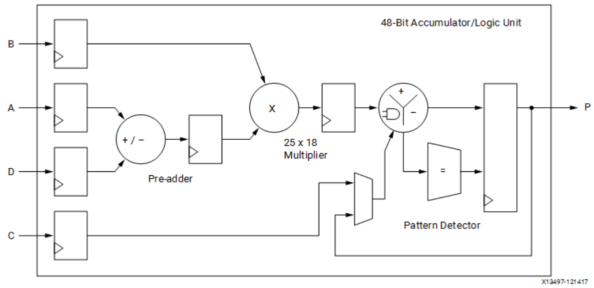
High-Level Synthesis Basics (2/2)

- HLS phases
  - Scheduling
  - Binding
    - Determines hardware resource to use
  - Control logic extraction
    - Create a FSM (Finite State Machine)

```
void foo(int in[3], char a, char b, char c, int out[3]) {
    int x,y;
    for(int i = 0; i < 3; i++) {
        x = in[i];
        y = a*x + b + c;
        out[i] = y;
    }
}
```



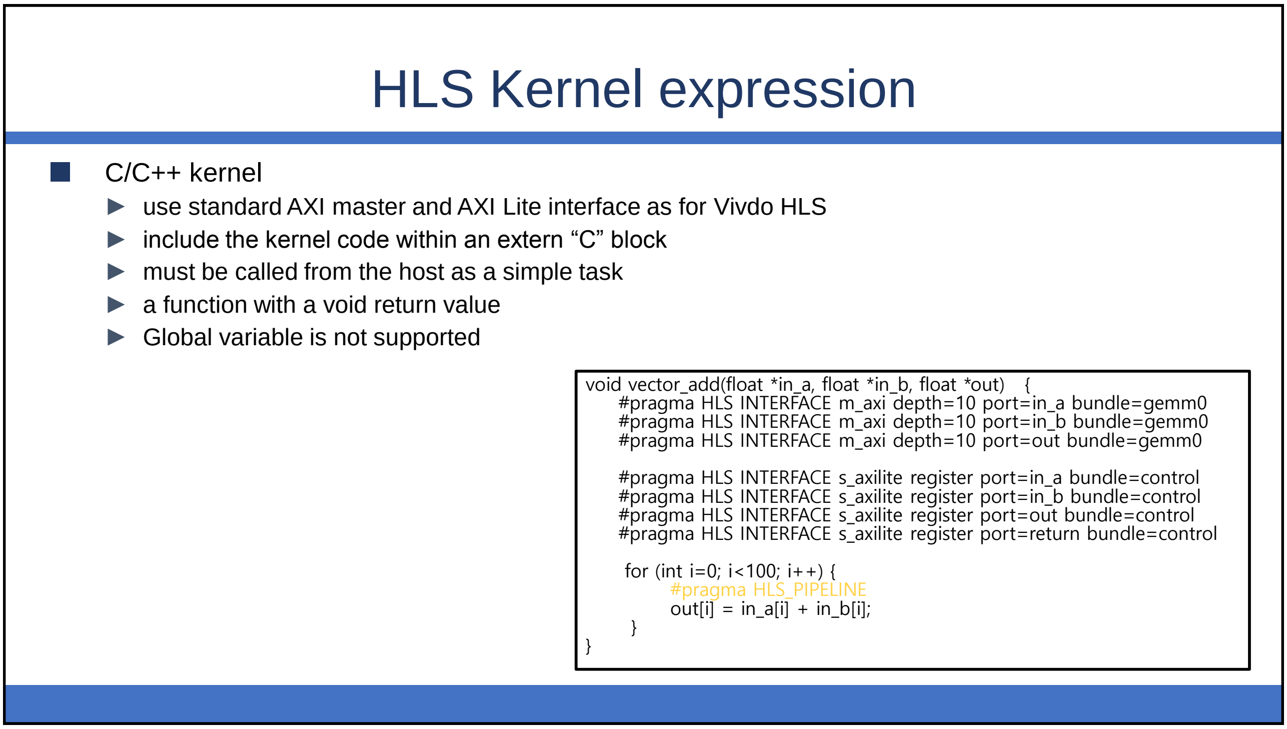
Xilinx DSP48 block



- The DSP48 block is an arithmetic logic unit (ALU) embedded into the fabric of the FPGA and is composed of a chain of three different blocks.
- P=Bx(A+D)+C
- P +=Bx(A+D)

Vivado HLS Pragmas by Type

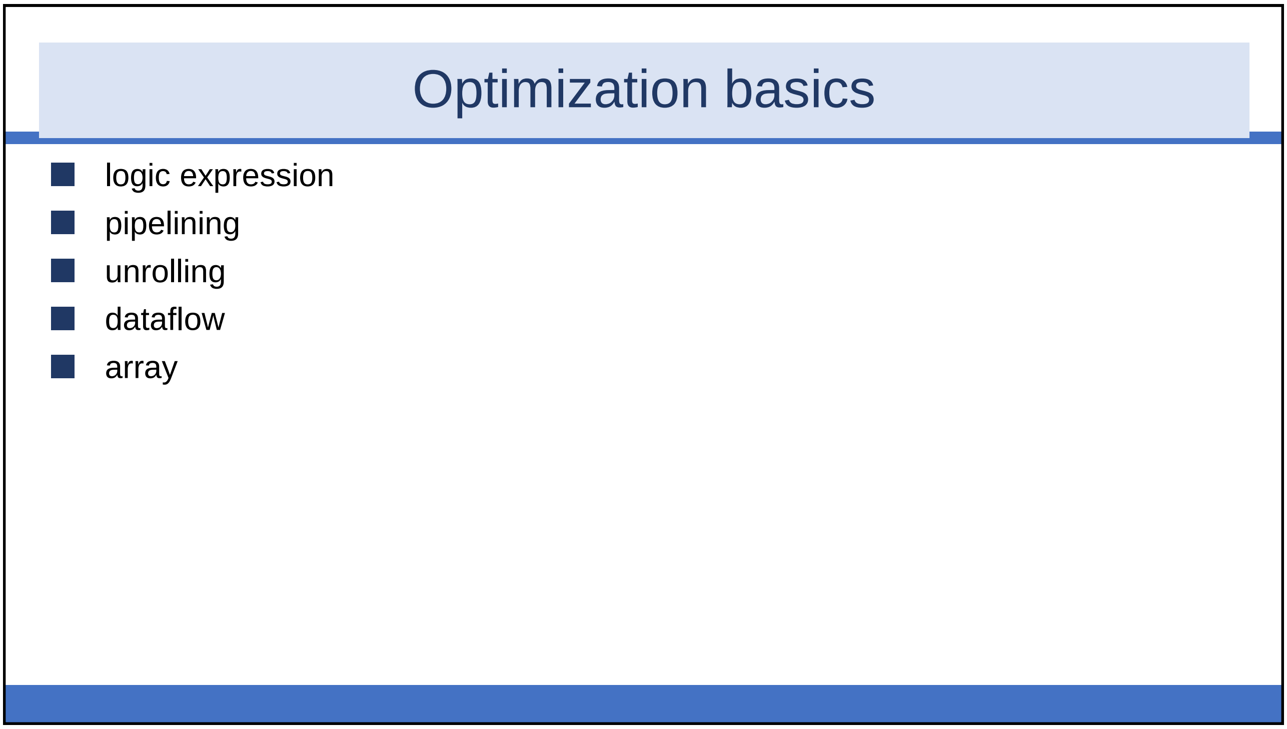
Type	Attribute	Type	Attribute
Kernel operation	pragma HLS allocation pragma HLS expression_balance pragma HLS latency	Kernel operation	pragma HLS reset pragma HLS resource pragma HLS stable
Function inlining	pragma HLS inline pragma HLS function_instantiate	Loop unrolling	pragma HLS unroll pragma HLS dependence
Interface synthesis	pragma HLS interface	Loop optimization	pragma HLS loop_flatten pragma HLS loop_merge pragma HLS loop_tripcount
Task-level pipeline	pragma HLS dataflow pragma HLS stream	Array optimization	pragma HLS array_map pragma HLS array_partition pragma HLS array_reshap
Pipeline	pragma HLS pipeline pragma HLS occurrence	Structure packing	pragma HLS data_pack



HLS Kernel expression

- C/C++ kernel
  - use standard AXI master and AXI Lite interface as for Vivdo HLS
  - include the kernel code within an extern “C” block
  - must be called from the host as a simple task
  - a function with a void return value
  - Global variable is not supported

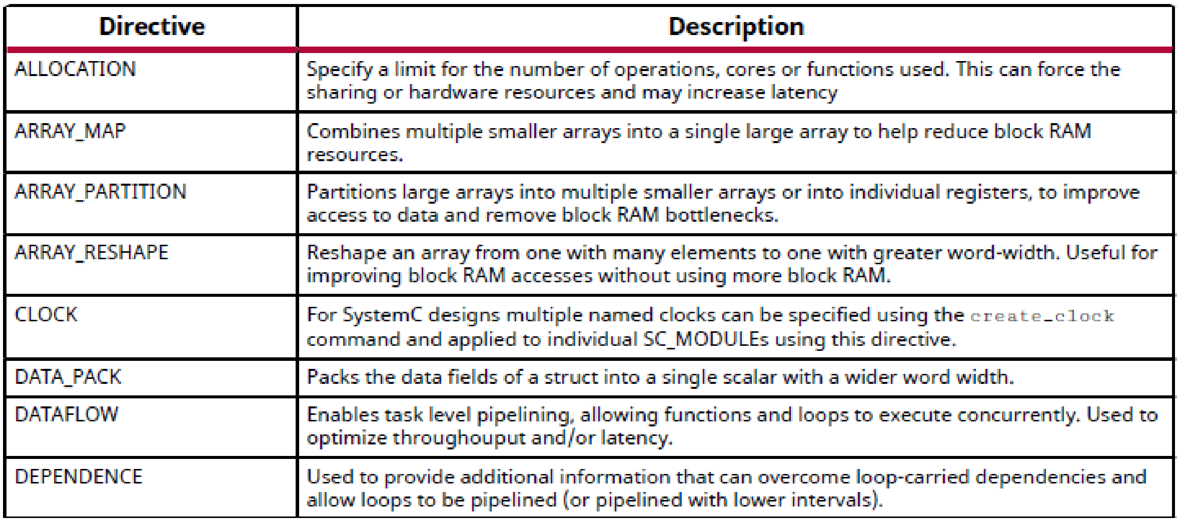
```
void vector_add(float *in_a, float *in_b, float *out)   {
     #pragma HLS INTERFACE m_axi depth=10 port=in_a bundle=gemm0
     #pragma HLS INTERFACE m_axi depth=10 port=in_b bundle=gemm0
     #pragma HLS INTERFACE m_axi depth=10 port=out bundle=gemm0

     #pragma HLS INTERFACE s_axilite register port=in_a bundle=control
     #pragma HLS INTERFACE s_axilite register port=in_b bundle=control
     #pragma HLS INTERFACE s_axilite register port=out bundle=control
     #pragma HLS INTERFACE s_axilite register port=return bundle=control

      for (int i=0; i<100; i++) {
             #pragma HLS_PIPELINE
             out[i] = in_a[i] + in_b[i];
       }
}
```



Optimization basics

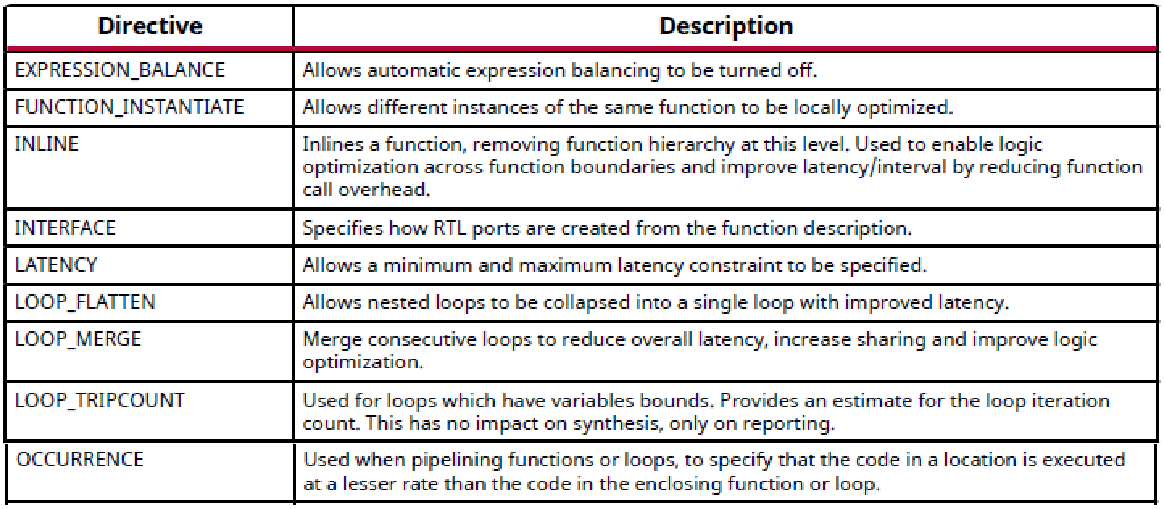
- logic expression
- pipelining
- unrolling
- dataflow
- array

Vivado HLS optimization directives (1/3)



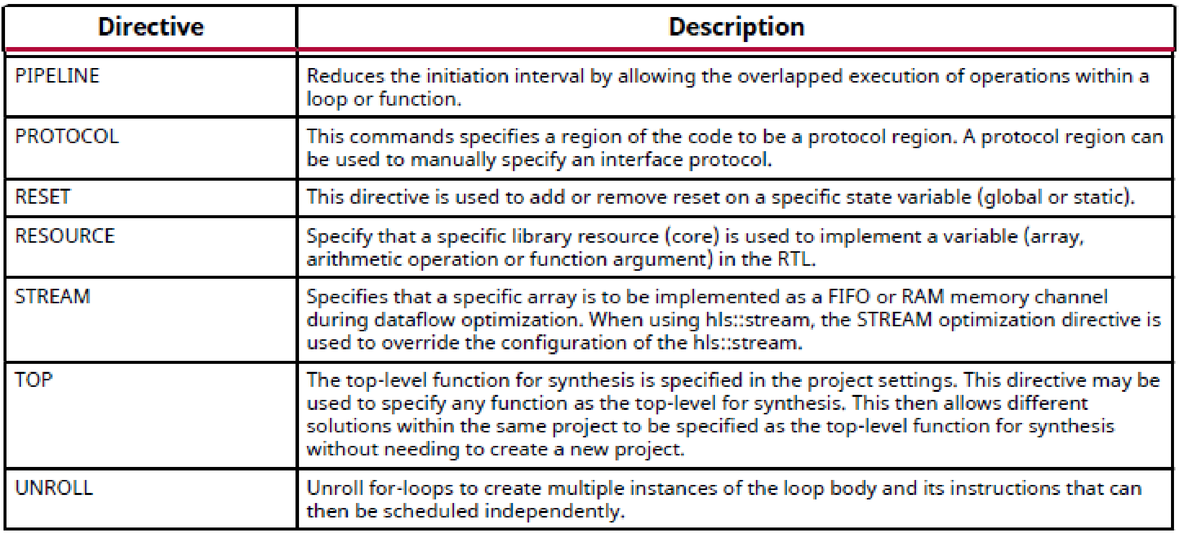
Directive	Description
ALLOCATION	Specify a limit for the number of operations, cores or functions used. This can force the sharing or hardware resources and may increase latency
ARRAY_MAP	Combines multiple smaller arrays into a single large array to help reduce block RAM resources.
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
ARRAY_RESHAPE	Reshape an array from one with many elements to one with greater word-width. Useful for improving block RAM accesses without using more block RAM.
CLOCK	For SystemC designs multiple named clocks can be specified using the `create_clock` command and applied to individual SC_MODULEs using this directive.
DATA_PACK	Packs the data fields of a struct into a single scalar with a wider word width.
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to optimize throughouput and/or latency.
DEPENDENCE	Used to provide additional information that can overcome loop-carried dependencies and allow loops to be pipelined (or pipelined with lower intervals).

Vivado HLS optimization directives (2/3)



Directive	Description
EXPRESSION_BALANCE	Allows automatic expression balancing to be turned off.
FUNCTION_INSTANTIATE	Allows different instances of the same function to be locally optimized.
INLINE	Inlines a function, removing function hierarchy at this level. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
INTERFACE	Specifies how RTL ports are created from the function description.
LATENCY	Allows a minimum and maximum latency constraint to be specified.
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop with improved latency.
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.
LOOP_TRIPCOUNT	Used for loops which have variables bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.
OCCURRENCE	Used when pipelining functions or loops, to specify that the code in a location is executed at a lesser rate than the code in the enclosing function or loop.

Vivado HLS optimization directives (3/3)

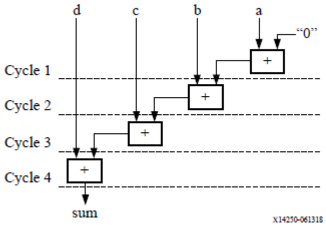


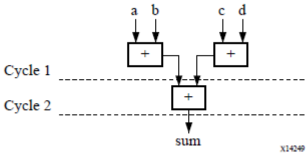
Directive	Description
PIPELINE	Reduces the initiation interval by allowing the overlapped execution of operations within a loop or function.
PROTOCOL	This commands specifies a region of the code to be a protocol region. A protocol region can be used to manually specify an interface protocol.
RESET	This directive is used to add or remove reset on a specific state variable (global or static).
RESOURCE	Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL.
STREAM	Specifies that a specific array is to be implemented as a FIFO or RAM memory channel during dataflow optimization. When using hls::stream, the STREAM optimization directive is used to override the configuration of the hls::stream.
TOP	The top-level function for synthesis is specified in the project settings. This directive may be used to specify any function as the top-level for synthesis. This then allows different solutions within the same project to be specified as the top-level function for synthesis without needing to create a new project.
UNROLL	Unroll for-loops to create multiple instances of the loop body and its instructions that can then be scheduled independently.

Optimization basics: logic expression

- Expression balancing rearranges operators to construct a balanced tree and reduce latency.
  - By default, Vivado HLS does not perform the ***EXPRESSION_BALANCE*** optimization for operations of type float or double.
  - It is default for integer operation.

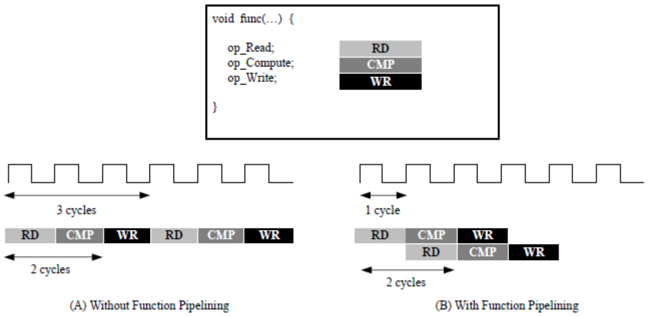
```
data_t foo_top (data_t a, data_t b, data_t c, data_t d) {
    data_t sum=0;
    sum += a;
    sum += b;
    sum += c;
    sum += d;
    return sum;
}
```



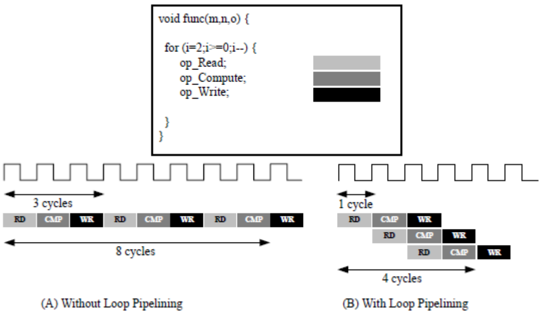


Optimization basics: pipelining

- Function pipelining
- Loop pipelining



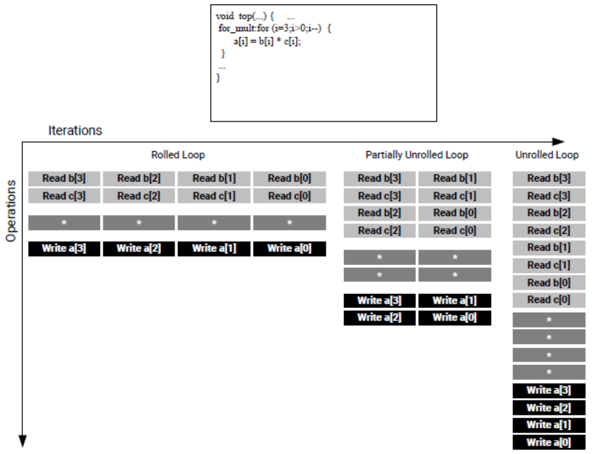
```
void  func(…) {

    op_Read;
    op_Compute;
    op_Write;

}
```



```
void func(m,n,o) {

  for (i=2;i>=0;i--) {
     op_Read;
     op_Compute;
     op_Write;

  }
}
```

Optimization basics: loop unrolling

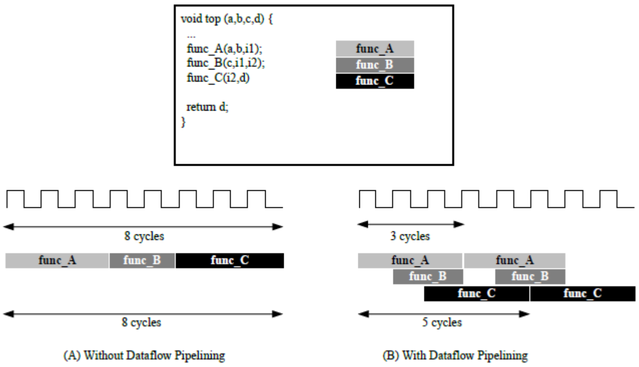
- To improve pipelining
- By default, loops are kept rolled in Vivado HLS.
  - These rolled loops generate a hardware resource which is used by each iteration of the loop.



```
void  top(...) {     ...
 for_mult:for (i=3;i>0;i--)  {
      a[i] = b[i] * c[i];
  }
 ...
}
```

Optimization basics: dataflow

- Task level parallelism, similar to function pipelining



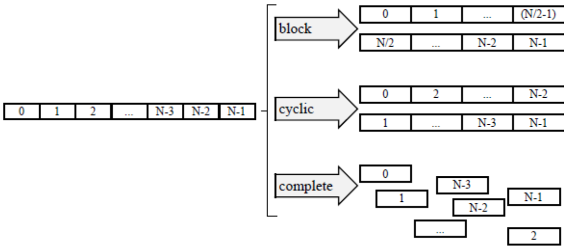
```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

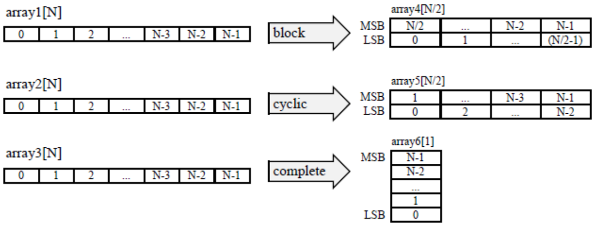
(A) Without Dataflow Pipelining

(B) With Dataflow Pipelining

Optimization basics: array

- The ARRAY_PARTITION directive to improve pipelining
- The ARRAY_RESHAPE directive allows more data to be accessed in a single clock cycle.





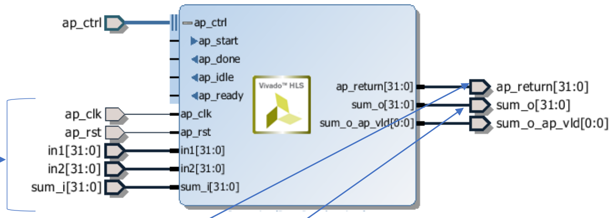
Interfaces

- In an RTL design these same input and output operations must be performed through a port in the design interface and typically operates using **a specific I/O (input-output) protocol**.
  - *Interface Synthesis* based on industry standard interface
    - clock, reset, block-level protocol, port-level protocol
  - *Manual interface specification* described in the source code
    - any arbitrary I/O protocol
      - Through SystemC design or C/C++ design

The top-level function becomes the top level of the RTL design after synthesis. Sub-functions are synthesized into blocks in the RTL design.

Interface synthesis

- The **arguments** (or parameters) of the function at top-level are synthesized into RTL port (*port-level interface protocol*) in addition to *clock*, *reset*, *block-level interface protocol*.



```
dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {
    dout_t temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;
    return temp;
}
```

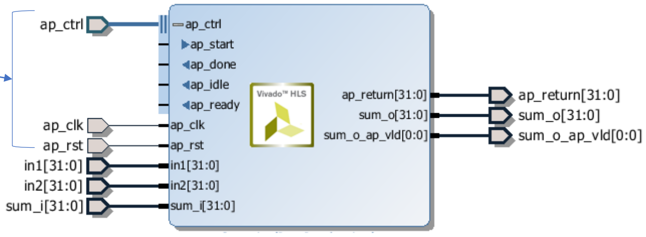
- Two pass-by-value inputs: in1, in2
- A pointer to read from and write to: sum
- A function return (ap_return)

Interface synthesis

- Clock and reset ports
  - ap_clk (rising edge synchronized)
  - ap_rst (active high)
- Block-level interface protocol (ap_ctrl_hs)
  - ap_start: start processing data
  - ap_done: completed operation (for ap_return)
  - ap_idle: idle state
  - ap_ready: ready to accept new inputs
- Function argument for both input (read from) and output (write to)
  - split into separate input and output ports
- Port-level interface protocol (without specific interface protocol)
  - input pass-by-value arguments
    - simple wire ports with no associated handshaking signals
  - input pointer arguments
    - the same as input pass-by-value arguments
  - output pointer arguments
    - simple wire ports with data valid signal
- Function return value
  - an output port: ap_return
  - a valid signal: ap_done
- The return value to the top-level function cannot be a pointer.

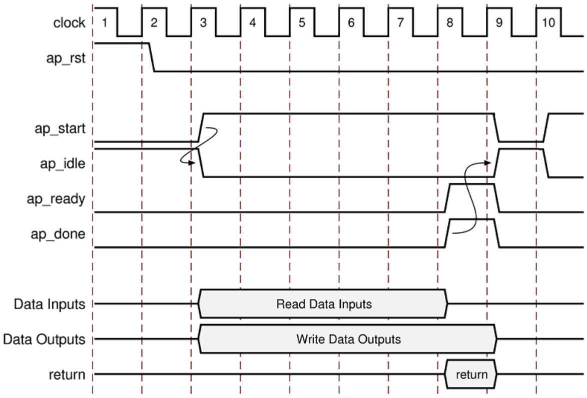
Interface synthesis: block-level interface protocol

- ap_ctrl_hs (default)
- ap_ctrl_chain
  - ap_ctl_hs with ap_continue for input
- ap_control_none



Interface synthesis: block-level interface protocol

- **ap_ctrl_hs**



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

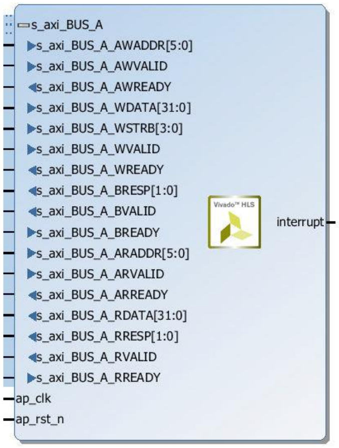
Interface synthesis: port-level interface protocol

- AXI4 interfaces
  - AXI4-Stream (axis)
    - only for input or output arguments, but not for input/output arguments
  - AXI4-Lite (s_axilite)
    - for any type of arguments except arrays.
    - can be grouped multiple arguments into the same AXI4-Lite interface
  - AXI4 Master (m_axi)
    - only for arrays and pointers (and references in C++).
    - can be grouped multiple arguments into the same AXI interface
- No I/O protocol interfaces
  - ap_none
  - ap_stable
- Wire handshakes interfaces
  - ap_hs
  - ap_vld
  - ap_ack
  - ap_ovld
- Memory interfaces
  - ap_memory
  - bram

Interface synthesis: port-level interface protocol

- AXI4-Lite interface
  - example
    - group multiple ports into the same AXI4-Lite interface using ‘bundle’
      - by default without ‘bundle’, all AXI4-Lite interfaces grouped into the same default bundled.

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



Interface synthesis: port-level interface protocol

- Typical control register layout for control through AXI4-Lite port.

```
// WAIT FOR READY
    ap_addr = ADDR_CSR; // 0x0000_0000
    while (1) {
        MEM_READ(ap_addr, ap_idle_r);
        ap_idle = (ap_idle_r >> 2) && 0x1;
        if (ap_idle)
            break;
    }
```

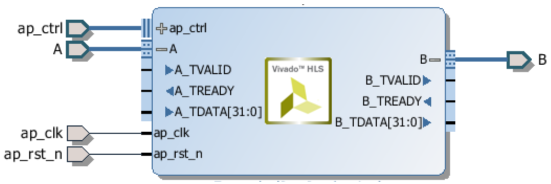
```
// LET GO AND WAIT FOR COMPLETION
    ap_addr = ADDR_CSR;
    ap_data = 0x1;
    MEM_WRITE(ap_addr, ap_data);//Start
    while (1) {
        MEM_READ(ap_addr, ap_done_r);
        ap_done = (ap_done_r >> 1) && 0x1;
        if (ap_done) break;
    }
```

```
// 0x00 : Control signals
//        bit 0  - ap_start (Read/Write/COH)
//        bit 1  - ap_done (Read/COR)
//        bit 2  - ap_idle (Read)
//        bit 3  - ap_ready (Read)
//        bit 7  - auto_restart (Read/Write)
//        others - reserved
// 0x04 : Global Interrupt Enable Register
//        bit 0  - Global Interrupt Enable (Read/Write)
//        others - reserved
// 0x08 : IP Interrupt Enable Register (Read/Write)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x0c : IP Interrupt Status Register (Read/TOW)
//        bit 0  - Channel 0 (ap_done)
//        bit 1  - Channel 1 (ap_ready)
//        others - reserved
// 0x10 : Data signal of shared_mem
//        bit 31~0 - shared_mem[31:0] (Read/Write)
// 0x14 : reserved
```

Interface synthesis: port-level interface protocol

- AXI4-Stream interface

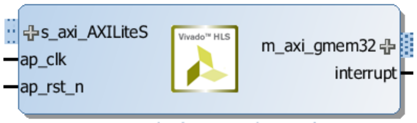
```
void example(int A[50], int B[50]) {
    #pragma HLS INTERFACE axis port=A
    #pragma HLS INTERFACE axis port=B
    int i;
    for(i = 0; i < 50; i++){
    B[i] = A[i] + 5;
}
```



Interface synthesis: port-level interface protocol

- AXI4-Master interface
  - Only for array or pointer arguments

```
void example(volatile int *a){
#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return
//Port a is assigned to an AXI4 master interface
int i; int buff[50];
//memcpy creates a burst access to memory
memcpy(buff,(const int*)a,50*sizeof(int));
for(i=0; i < 50; i++){
buff[i] = buff[i] + 100;
}
memcpy((int *)a,buff,50*sizeof(int));
}
```



High-Level Synthesis C Libraries

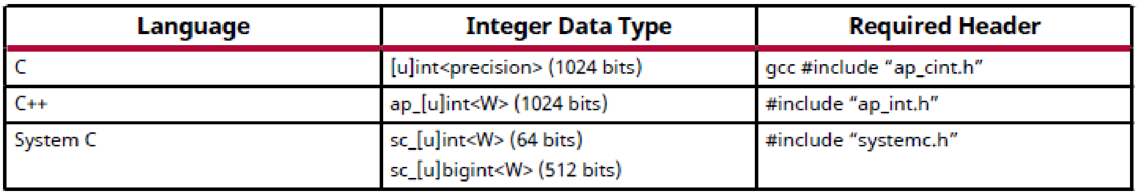
- Arbitrary Precision Data Types Library
- HLS Stream Library
- HLS Math Library
- HLS Video Library
- HLS IP Library
- HLS Linear Algebra Library
- HLS DSP Library
- HLS SQL Library

- C libraries can be synthesized to RTL

Arbitrary integer precision data types

- C-based native data types are on 8-bit boundaries (8, 16, 32, 64 bits).
- RTL buses (corresponding to hardware) support arbitrary lengths.

- Vivado HLS integer data types ($XILINX_VIVADO/include/)
  - ▶ “ap_cint.h” for C and bit-width can be 1 to 1024, e.g., in7, uint123, ...
  - ▶ “ap_int.h:” for C++ and bit-with can be any e.g., ap_int<7>, ap_uint<123>

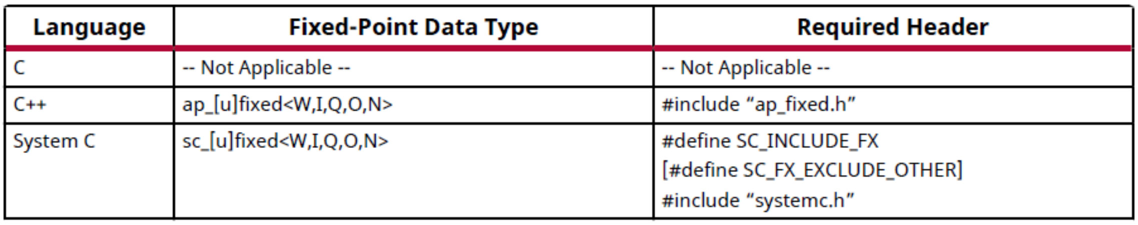


Language	Integer Data Type	Required Header
C	[u]int<precision> (1024 bits)	gcc #include “ap_cint.h”
C++	ap_[u]int<W> (1024 bits)	#include “ap_int.h”
System C	sc_[u]int<W> (64 bits) sc_[u]bigint<W> (512 bits)	#include “systemc.h”

Arbitrary precision fixed-point data types (1/2)

- Fixed-point data type manages the value of real (non-integer) numbers within the boundaries of a specified total width and integer width.

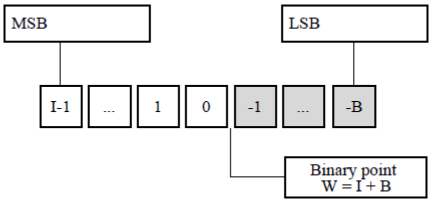
- Vivado HLS fixed-point data types ($XILINX_VIVADO/include/)
  - “ap_fixed.h:” for C++ and bit-with can be any e.g., ap_fixed<18,6,AP_RND>, ap_ufixed<...>

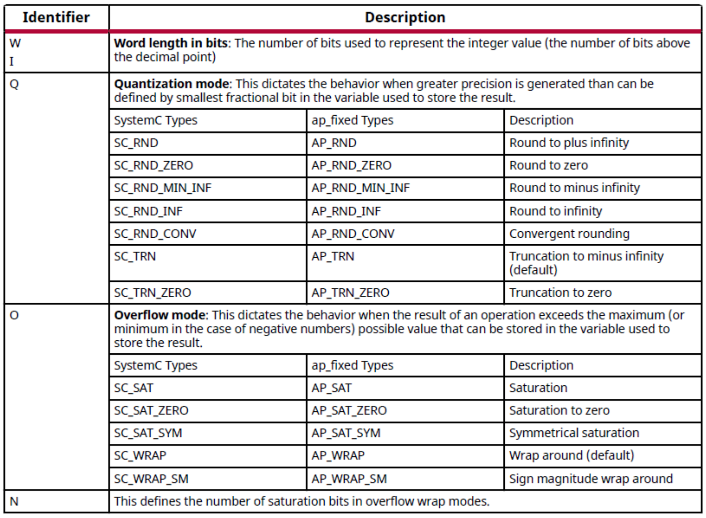


Language	Fixed-Point Data Type	Required Header
C	-- Not Applicable --	-- Not Applicable --
C++	ap_[u]fixed<W,I,Q,O,N>	#include “ap_fixed.h”
System C	sc_[u]fixed<W,I,Q,O,N>	#define SC_INCLUDE_FX [#define SC_FX_EXCLUDE_OTHER] #include “systemc.h”

Arbitrary precision fixed-point data types (2/2)

- Word length in bits:***W=I+B***
  - ***ap_[u]fixed<W,I,Q,O,N>***





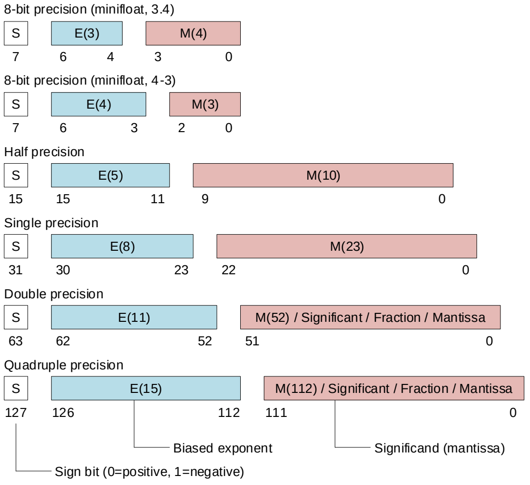
Identifier	Description		
W I	**Word length in bits**: The number of bits used to represent the integer value (the number of bits above the decimal point)		
Q	**Quantization mode**: This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result.		
	SystemC Types	ap_fixed Types	Description
	SC_RND	AP_RND	Round to plus infinity
	SC_RND_ZERO	AP_RND_ZERO	Round to zero
	SC_RND_MIN_INF	AP_RND_MIN_INF	Round to minus infinity
	SC_RND_INF	AP_RND_INF	Round to infinity
	SC_RND_CONV	AP_RND_CONV	Convergent rounding
	SC_TRN	AP_TRN	Truncation to minus infinity (default)
	SC_TRN_ZERO	AP_TRN_ZERO	Truncation to zero
O	**Overflow mode**: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result.		
	SystemC Types	ap_fixed Types	Description
	SC_SAT	AP_SAT	Saturation
	SC_SAT_ZERO	AP_SAT_ZERO	Saturation to zero
	SC_SAT_SYM	AP_SAT_SYM	Symmetrical saturation
	SC_WRAP	AP_WRAP	Wrap around (default)
	SC_WRAP_SM	AP_WRAP_SM	Sign magnitude wrap around
N	This defines the number of saturation bits in overflow wrap modes.		

HLS stream library

- Streaming data is a type of data transfer in which data samples are sent in sequential order starting from the first sample.
  - Streaming requires no address management.
  - *The* hls::stream *class is only used in C++ designs. Array of streams is not supported*
- #include “hls_stream.h”

HLS math library

- The Vivado HLS Math Library (hls_math.h) provides support for the synthesis of the standard C (math.h) and C++ (cmath.h) libraries and is automatically used to specify the math operations during synthesis.
- The support includes floating point (single-precision: type ***float***), double precision: type ***double*** and half-precision: type ***half***) for all functions and fixed-point: type ***ap_[u]fixed<W,I,Q,O,N>*** support for some functions.
  - half_func() for half-precision only, e.g., half_sin()
  - funcf() for single-precision only, e.g., sinf()



HLS math library

- For floating point case
  - Use ‘match.h’ or ‘cmath.” or Vivado specific ‘hls_math.h”
  - Vivado HLS math library ($XILINX_VIVADO/include/)
    - “hls_math.h:” for C++
- For fixed-point case
  - support “ap_[u]fixed” and “ap_[u]int” with following bit-width specification
    - ap_fixed<W,I> where I<=33 and W-I<=32
    - ap_ufixed<W,I> where I<=32 and W-I<=32
    - ap_int<I> where I<=33
    - ap_uint<I> where I<=32

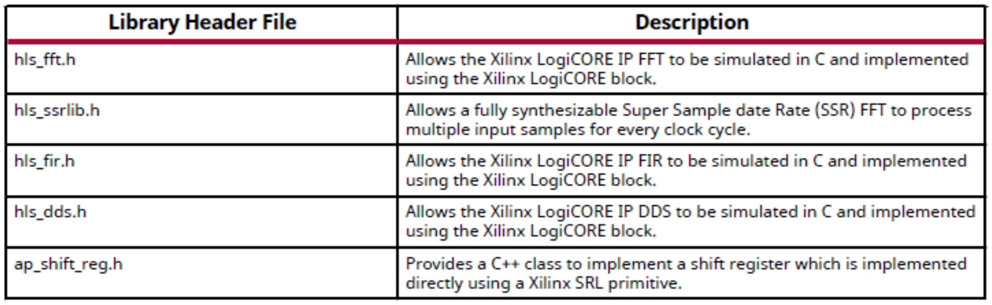
```
#include "hls_math.h"
#include "ap_fixed.h“

ap_fixed<32,2> my_input, my_output;
my_input = 24.675;
my_output = sin(my_input);
```

HLS video library

- https://github.com/Xilinx/xfopencv
- The xfOpenCV library is a set of 60+ kernels, optimized for Xilinx FPGAs and SoCs, based on the OpenCV computer vision library.

HLS IP Library

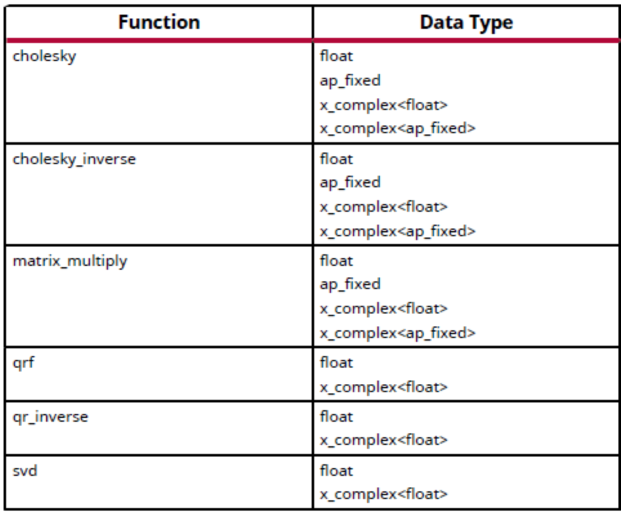
- Vivado HLS provides C++ libraries to implement a number of Xilinx IP blocks.



Library Header File	Description
hls_fft.h	Allows the Xilinx LogiCORE IP FFT to be simulated in C and implemented using the Xilinx LogiCORE block.
hls_ssrlib.h	Allows a fully synthesizable Super Sample date Rate (SSR) FFT to process multiple input samples for every clock cycle.
hls_fir.h	Allows the Xilinx LogiCORE IP FIR to be simulated in C and implemented using the Xilinx LogiCORE block.
hls_dds.h	Allows the Xilinx LogiCORE IP DDS to be simulated in C and implemented using the Xilinx LogiCORE block.
ap_shift_reg.h	Provides a C++ class to implement a shift register which is implemented directly using a Xilinx SRL primitive.

HLS Linear Algebra Library

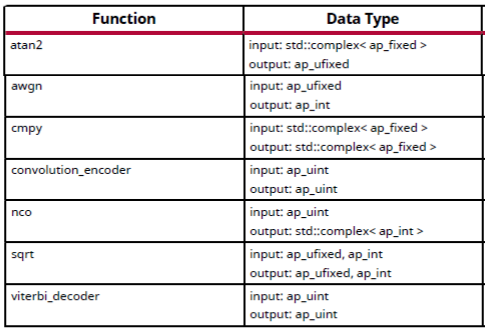
- The HLS Linear Algebra Library provides a number of commonly used C++ linear algebra functions.
  - The functions in the HLS Linear Algebra Library all use two-dimensional arrays to represent matrices and support single-precision float for real and complex data and ap_fixed with limited case.
  - #include "hls_linear_algebra.h"



Function	Data Type
cholesky	float ap_fixed x_complex<float> x_complex<ap_fixed>
cholesky_inverse	float ap_fixed x_complex<float> x_complex<ap_fixed>
matrix_multiply	float ap_fixed x_complex<float> x_complex<ap_fixed>
qrf	float x_complex<float>
qr_inverse	float x_complex<float>
svd	float x_complex<float>

HLS DSP Library

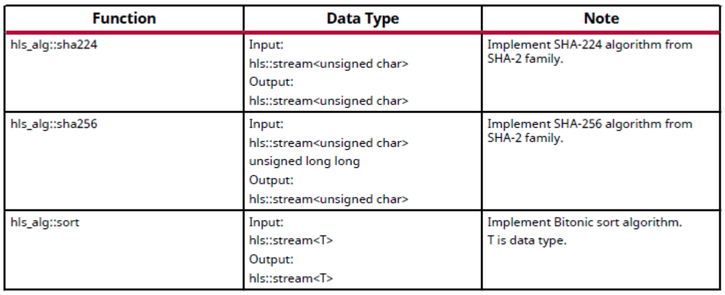
- The HLS DSP library contains building-block functions for DSP system modeling in C++ with an emphasis on functions used in SDR(Software Defined Radio) applications.
- #include <hls_dsp.h>



Function	Data Type
atan2	input: std::complex< ap_fixed > output: ap_ufixed
awgn	input: ap_ufixed output: ap_int
cmpy	input: std::complex< ap_fixed > output: std::complex< ap_fixed >
convolution_encoder	input: ap_uint output: ap_uint
nco	input: ap_uint output: std::complex< ap_int >
sqrt	input: ap_ufixed, ap_int output: ap_ufixed, ap_int
viterbi_decoder	input: ap_uint output: ap_uint

HLS SQL Library

- SQL (Structured Query Language) building-block functions in C++.
- #include <hls_alg.h>



Function	Data Type	Note
hls_alg::sha224	Input: hls::stream<unsigned char> Output: hls::stream<unsigned char>	Implement SHA-224 algorithm from SHA-2 family.
hls_alg::sha256	Input: hls::stream<unsigned char> unsigned long long Output: hls::stream<unsigned char>	Implement SHA-256 algorithm from SHA-2 family.
hls_alg::sort	Input: hls::stream<T> Output: hls::stream<T>	Implement Bitonic sort algorithm. T is data type.

High-level synthesis coding styles

- Do not use system calls
- dynamic memory

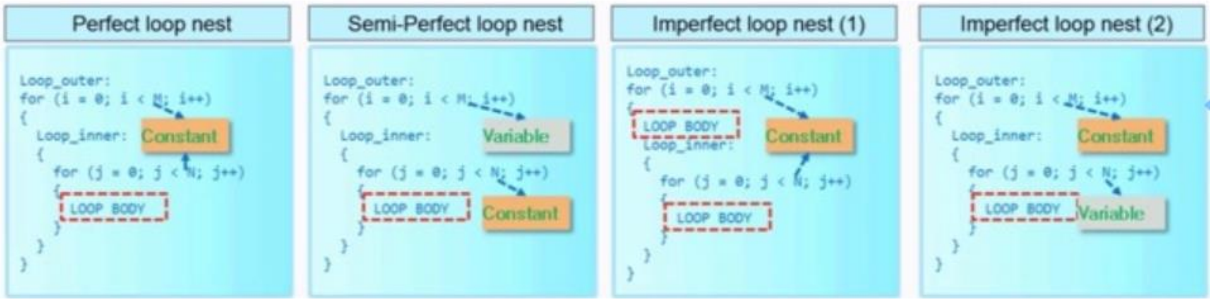
Do not use system calls

- Use __SYNTHESIS__ macro (note double _) to exclude un-supported statements (non-synthesizable code)
  - #ifndef __SYNTHESIS__
  - any code
  - #endif
- HLS does not support common system calls
  - automatically ignored (display only purposes): printf(), fprintf()
  - should be removed from the function before synthesis: getc(), time(), sleep(), ..
- Memory allocation system calls must be removed from the design code before synthesis.
  - malloc(), alloc(), free()
- Recursive functions
- Standard templated libraries (STL) since it usually uses function recursion and dynamic memory allocation

High-level synthesis: loops

Types of loop

- Perfect, semi-perfect, imperfect loop



Perfect loop nest

```
Loop_outer:
for (i = 0; i < M; i++)
{
  Loop_inner:
  {
    for (j = 0; j < N; j++)
    {
      LOOP BODY
    }
  }
}
```

Semi-Perfect loop nest

```
Loop_outer:
for (i = 0; i < M; i++)
{
  Loop_inner:
  {
    for (j = 0; j < N; j++)
    {
      LOOP BODY
    }
  }
}
```

Imperfect loop nest (1)

```
Loop_outer:
for (i = 0; i < M; i++)
{
  LOOP BODY
  Loop_inner:
  {
    for (j = 0; j < N; j++)
    {
      LOOP BODY
    }
  }
}
```

Imperfect loop nest (2)

```
Loop_outer:
for (i = 0; i < M; i++)
{
  Loop_inner:
  {
    for (j = 0; j < N; j++)
    {
      LOOP BODY
    }
  }
}
```

References

- *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)