Coding Considerations

XILINX.

Objectives

- **After completing this module, you will be able to:**
  - List language support in Vivado HLS
  - State when a construct is not synthesizable
  - List unsupported constructs
  - Describe multi-access pointer and restriction imposed on using it
  - Describe how structs and pointers are handled in Vivado HLS
  - Generate efficient hardware from C sources using stream and shift register classes
  - List supported functions in OpenCV library

Outline

- Language Support
- Pointers
- Coding Considerations and IO
- Streams
- Libraries Support
  - FFT, FIR, Shift Registers and Linear Algebra
  - OpenCV
- Summary

Comprehensive C Support

- **A Complete C Validation & Verification Environment**
  - Vivado HLS supports complete bit-accurate validation of the C model
  - Vivado HLS provides a productive C-RTL co-simulation verification solution
- **Vivado HLS supports C, C++ and SystemC**
  - Functions can be written in any version of C
  - Wide support for coding constructs in all three variants of C
    - It’s easier to discuss what’s not supported than what is
- **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

Comprehensive C Support

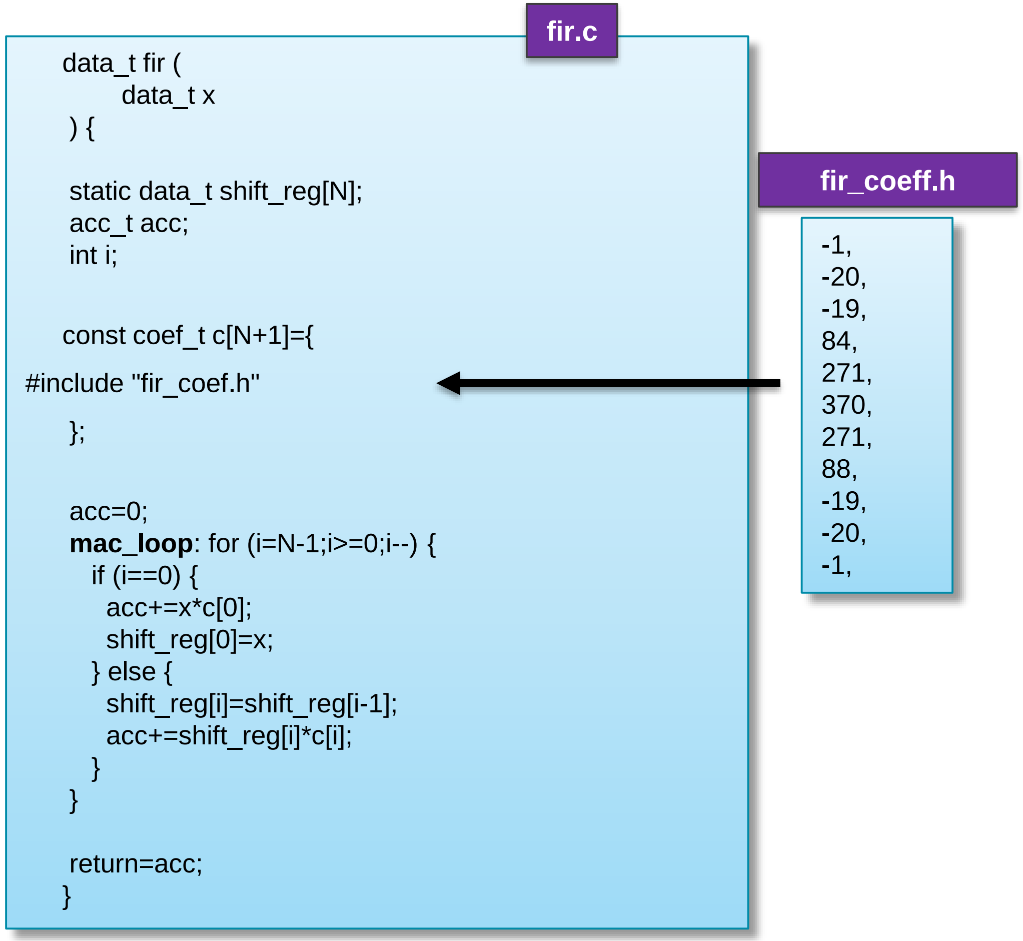
- **Pointers based applications**
  - Multi-access pointer issues and streams
- **Streaming and shift registering applications**
  - Generate efficient hardware from C sources using stream and shift register classes
- **Support for OpenCV functions**
  - Enable migration of OpenCV designs into Xilinx FPGA
  - Libraries target real-time full HD video processing

C, C++ and SystemC Support

- **The vast majority of C, C++ and SystemC is supported**
  - Provided it is statically defined at **compile time**
  - The C constructs must be of a fixed or bounded size and must be unamiguous
  - If it’s not defined until **run time**, it won’t be synthesizable
  - System calls to OS are not synthesizable
- **Any of the three variants of C can be used**
  - If C is used, Vivado HLS expects the file extensions to be .c
  - For C++ and SystemC it expects file extensions .cpp
- **Let’s look at an example using a FIR**
  - Single data input x
  - Coefficients are stored in a ROM
  - A single output: function return

FIR Filter with C

- **C coding style**
  - The top-level function defines the IO ports
  - If required, sub-functions can be added
- **Storage**
  - Static: required for anything which must hold it’s value between invocations
    - A C simulation will confirm what is required to be a static
    - Will become a register or memory
    - Other variables may become storage, depending on the implementation
  - Const: ensures the values are seen as constants which never changed
- **FIR details**
  - Coefficients are loaded from a file
    - Use the “const” keyword to ensure the data is seen as constant and a ROM is used



fir.c

```
data_t fir (
        data_t x
 ) {

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


const coef_t c[N+1]={

#include "fir_coef.h"

 };


 acc=0;
 mac_loop: for (i=N-1;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];
    }
 }

 return=acc;
}
```

fir_coeff.h

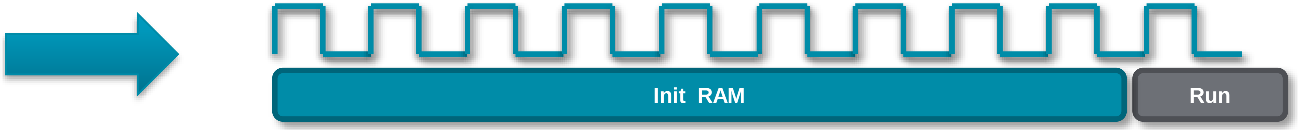
```
-1,
-20,
-19,
84,
271,
370,
271,
88,
-19,
-20,
-1,
```

Arrays to RAMs : Initialization

- **Static and Const have great impact on array implementation**
  - Const, as shown on previous slide, implies a ROM
  - Static can impact how RAMs are initialized

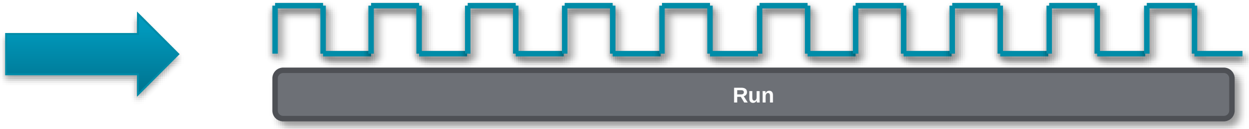
- **A typical coding style changes**

```
int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```



  - Implies coeff is set each time the function is executed
  - Resets in the RAM being explicitly loaded each transaction
    - Costs clock cycles
  - The array can be initialized as a static

```
static int coeff[8] = {4, -2, 8, -4, 10, 14, 10, -4, 8, -2, 4};
```



  - Statics are initialized at “start up” only
    - In the C, in the RTL via bitstream

A coefficient array does not need to be initialized every time but doing that costs “nothing” in software, unlike hardware

FIR Filter with C++

> C++ Coding Style

- Top-level defines RTL IO
- A class defines the functionality
  - Additional classes or sub-functions can be added
- The class is instantiated in the top-level function
  - Methods can be used

fir.h

```
#include “types.h”
template<class coef_T, class data_T, class acc_T>
class CFir
  {
protected:
  static const coef_T c[];
  data_T shift_reg[N-1];
private:
public:
  data_T operator()(data_T x);
  template<class coef_TT, class data_TT, class acc_TT>

  };

data_t fir(data_t x);
```

CFir Class defined

fir.cpp

```
template<class coef_T, class data_T, class acc_T>
const coef_T CFir<coef_T, data_T, acc_T>::c[] =
 {
#include "fir_coeff.h"
   };

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;

    mac_loop: for (i=N-1;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];
       }
     }
  return acc;
 }

data_t fir(data_t x)
 {
   static CFir<coef_t, data_t, acc_t> fir1;

   return fir1(x);
 }
```

CFir functionality defined

fir_coeff.h

```
-1,
-20,
-19,
84,
271,
370,
271,
88,
-19,
-20,
-1,
```

Top-level function defines the IO Class CFIR is instantiated as fir1

FIR Filter with SystemC

- **SystemC Style**
  - The header defines the IO ports
  - The constructor calls the functions
    - Can be SC_CTHREAD sensitive on the clock (with optional reset)
    - Can be SC_METHOD sensitive on any signal
  - Multiple threads, methods and functions can be used

fir.h

```
SC_MODULE (fir) {
//-----------Input Ports--------------
sc_in    <bool>                          clk;
sc_in    <bool>                          reset;
sc_in    <sc_uint<8> >      x;
//-----------Output Ports--------------
sc_out   <sc_uint<8> >    y;
//------------Internal Variables--------
sc_uint<8>  acc, m, i;
sc_uint<8> shift_reg[N];
//-------------Code Starts Here---------
void go ();

SC_CTOR(fir) {
                        SC_CTHREAD (go, clk.pos());
                        reset_signal_is(reset, true);
                        }
};
```

IO defined

Clock sensitive thread (SC_CTHREAD)

SC_METHOD is also supported (combo)

SC_THREAD is not supported

fir.cpp

```
#include “fir.h"

const sc_int<8> c ={
#include "fir_coeff.h“
                                        };

void fir::go (void) {
                        if (reset.read() == 1) {
                                    acc=0;
                                    i=0;
                        }
                        wait ();
                        while (1) {
                               mac_loop: for (i=N-1;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;
wait();
                        }
}
```

fir_coeff.h

Reset behavior is defined in the code (optional)

Wait() allows for a break in the infinitive processing loop

XILINX

Unsupported Constructs: Overview

- **System calls and function pointers**
  - Dynamic memory allocation
    - malloc() & free()
  - Standard I/0 and file I/O operations
    - fprintf() / fscanf() etc.
  - System calls
    - time(), sleep() etc.
- **Data types**
  - Forward declared type
  - Recursive type definitions
    - Type contains members with the same type
- **Non-standard Pointers**
  - Pointer casting between general data types
    - OK with native integers types
  - If a double pointer is used in multiple functions, Vivado HLS will inline all the functions
    - Slower synthesis, may increase area & run time

Unsupported: Dynamic Memory Allocation

- **Dynamic memory allocation**
  - Not allowed since it requires the construction (or destruction) of hardware at runtime
  - *malloc*, *alloc*, *free* are **<u>not</u>** synthesizable
  - Similarly for constructor initialization

```
long long x = malloc (sizeof(long long));

int* arr = malloc (64 * sizeof(int));
```

  - Use persistent static variables and fixed-size arrays instead

```
static long long x;

int array[64];
```

- **Dynamic virtual function call**

```
Class A {
public:

  virtual void bar() {…};

};
```

```
void fun(A* a) {
    a->bar();
}

A* a = 0;

if (base) A= new A();
else A = new B();

foo(a);
```

Unsupported: System Calls

- **System calls**
  - Most of C system calls do not have hardware counterparts and are not synthesizable
    - printf(), getc(), time(), ...
  - Vivado HLS will ignore system calls
    - They can also be removed by the __SYNTHESIS__ macro
  - Good Practice:
    - Use Macro "__SYNTHESIS__" to remove code segment involving system calls from synthesis
    - Macro "__SYNTHESIS__" is automatically defined by the Vivado HLS during elaboration

Only read this code if macro __SYNTHESIS__ is not set

```
void foo (…]) {
   …
#ifndef __SYNTHESIS__
  Code will  be seen by simulation.
  But not synthesis.
#endif
   …
}
```

Unsupported: General Pointer Casting

- **Pointer reinterpretation**
  - Casting a pointer to a different type is not allowed in the general case

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

*(unsigned*)pair = -1U;
```

– Solution: assign values using the original type

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

pair.first = -1U;
pair.second = -1U;
```

- **Pointer casting <u>is allowed</u> between native C integer types**

```
int foo (int index,  int A[N]) {
  char* ptr;
  int i =0, result = 0;

  ptr = (char*)(&A[index]);


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

Unsupported: Recursive Functions

- **Avoid recursive functions**
  - Not synthesizable in general
    - The code re-entrance indirectly uses dynamic memory allocation

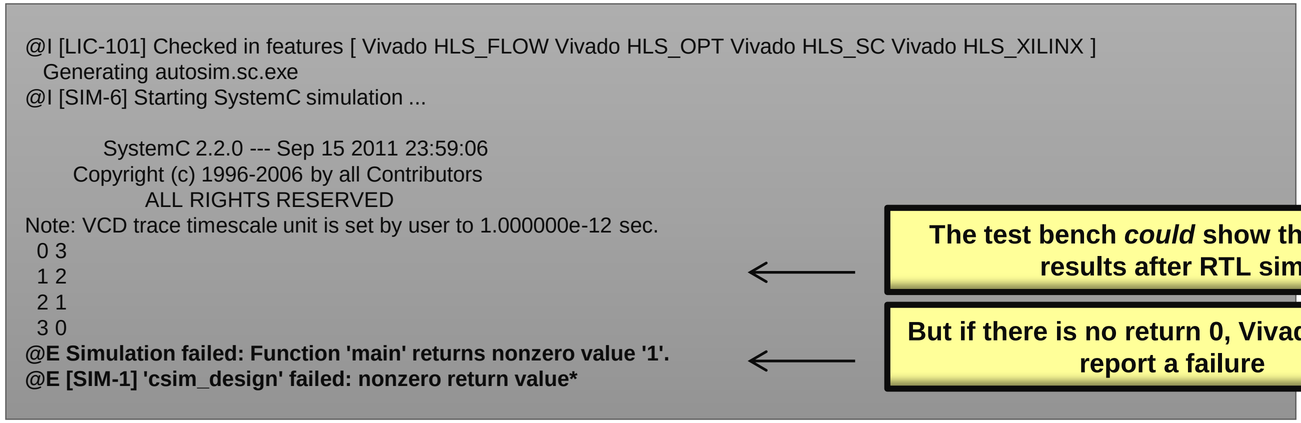
```
unsigned foo (unsigned n)
{
    if (n == 0 || n == 1) return 1;
    return (foo(n-2) + foo(n-1));
}
```

Not synthesizable: Endless recursion

- **Standard Template Libraries (STLs)**
  - Many are unbounded or use recursive functions
    - In general they cannot be synthesized
  - Re-code to create a local bounded model
    - Use arrays instead of vectors
    - Shifts instead of queues

Test Bench: Gotcha 1

- **Use a RETURN value**
  - RTL verification re-uses the C test bench to verify the RTL design
  - If the test bench does not return a 0, RTL sim will say it failed



```
@I [LIC-101] Checked in features [ Vivado HLS_FLOW Vivado HLS_OPT Vivado HLS_SC Vivado HLS_XILINX ]
   Generating autosim.sc.exe
@I [SIM-6] Starting SystemC simulation ...

            SystemC 2.2.0 --- Sep 15 2011 23:59:06
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED
Note: VCD trace timescale unit is set by user to 1.000000e-12 sec.
  0 3
  1 2
  2 1
  3 0
@E Simulation failed: Function 'main' returns nonzero value '1'.
@E [SIM-1] 'csim_design' failed: nonzero return value*
```

The test bench *could* show the correct results after RTL sim

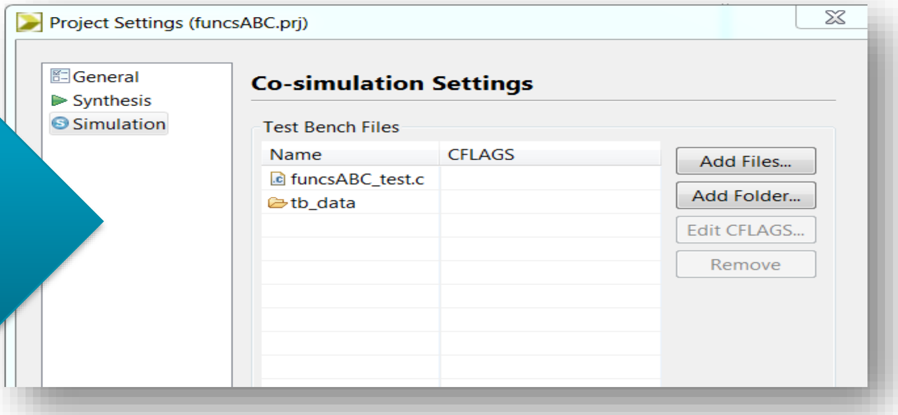
But if there is no return 0, Vivado HLS will report a failure

Test Bench : Gotcha 2

- **Include all files used by the test bench in Vivado HLS**
  - This test bench reads golden data from files

```
int main () {
   …
          // Get data from files
          fp=fopen("tb_data/inA.dat","r");
          for (i=0; i<N*NUM_TRANS; i++){
                   int tmp;
                   fscanf(fp, "%d", &tmp);
                   a_gold[i] = tmp;
          }
          fclose(fp);

          fp=fopen("tb_data/inB.dat","r");
          for (i=0; i<N*NUM_TRANS; i++){
                   int tmp;
                   fscanf(fp, "%d", &tmp);
                   b_gold[i] = tmp;
          }
          fclose(fp);
  …
  return ret;
}
```

Include file directory with the test bench



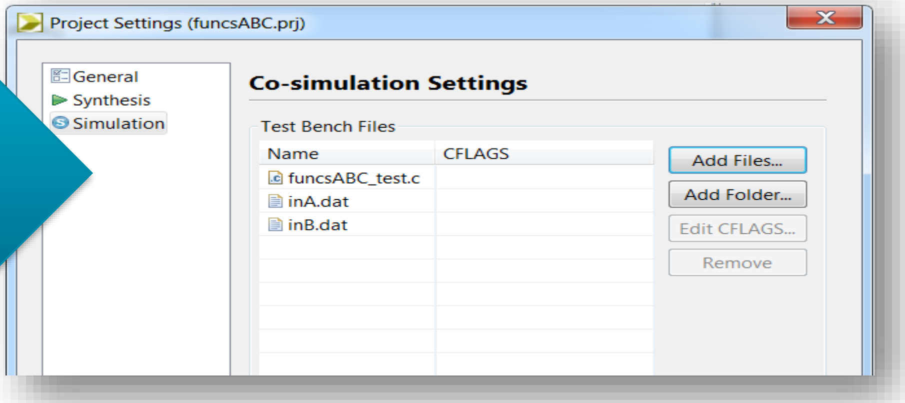
Project Settings (funcsABC.prj)

Co-simulation Settings

Test Bench Files

Name	CFLAGS
funcsABC_test.c	
tb_data	

Or include both files



Project Settings (funcsABC.prj)

Co-simulation Settings

Test Bench Files

Name	CFLAGS
funcsABC_test.c	
inA.dat	
inB.dat	

  - Include all files or RTL verification **will** fail

Pointers

Using Pointer

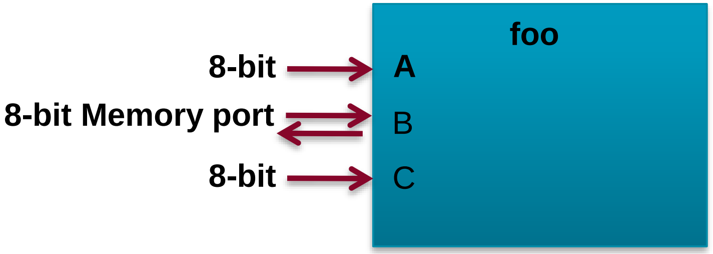
- **Structs as pointers**
  - Structs are implemented differently as pointers or pass-by-value
- **Pointer arithmetic**
  - Only supported as interface using ap_bus
- **Converting pointers using malloc**
  - Must be converted to fixed sized resources
- **Multi-access Pointers**
  - Must be marked as volatile or reads and writes will be optimized away
  - Cannot be rigorously validated with C simulation
  - Require a depth specification for RTL simulation
- **Arrays of Pointers**
  - When using arrays of pointers, each pointer must point to a scalar or a scalar array (not another pointer)

Pointer & Structs

- **Struct passed as a pointer**
  - When a struct is passed as a pointer
  - The data will be accessed as pass-by-reference

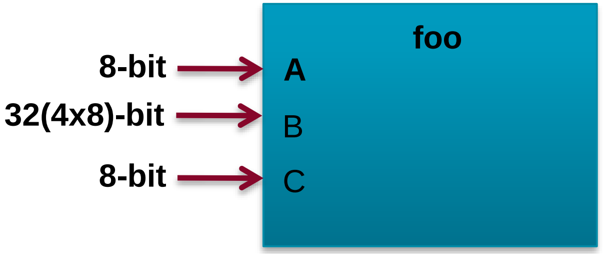
```
typedef struct{
        unsigned char A;
        unsigned char B[4];
        unsigned char C;
} my_data;

void foo(my_data *in);
```

Pointer



- **Struct passed by-value**
  - When a struct is passed directly and not as a pointer
  - The data will be accessed as pass-by-value

```
typedef struct{
        unsigned char A;
        unsigned char B[4];
        unsigned char C;
} my_data;

void foo(my_data in);
```



Struct Access

> **Accessing a struct: pass-by-reference**

```
void foo(my_data *a_in, my_data *b_out) {
        int i;

        b_out->A = a_in->A + 1;

        for(i=0; i <= 319; i++) {
                b_out->B[i] = a_in->B[i] +1 ;
        }

        b_out->C = a_in->C +1;
}
```

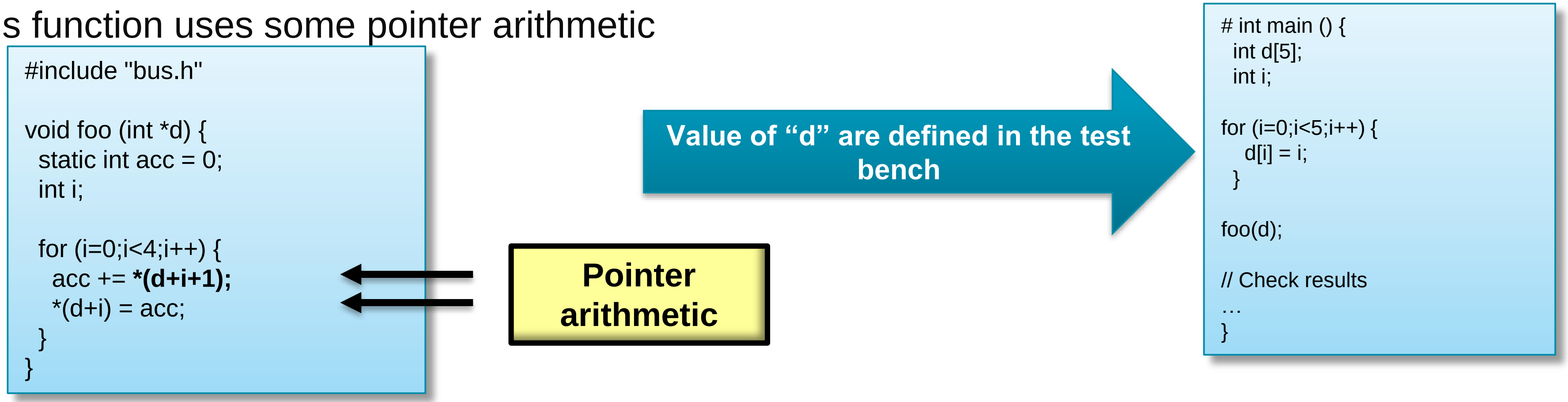
```
typedef struct {
        unsigned char A;
        unsigned char B[320];
        unsigned char C;
}my_data;
```

> **Accessing a struct: pass-by-value**

```
my_data foo(my_data a_in, my_data b_out) {
        int i;

        b_out.A = a_in.A+1;

        for(i=0; i <= N-1; i++) {
                b_out.B[i] = a_in.B[i]+1;
        }

        b_out.C = a_in.C+1;

        return b_out;
}
```

Pointer Arithmetic: One Interface

- **Pointer arithmetic**
  - This function uses some 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;
  }

foo(d);

// Check results
…
}
```

  - Argument can be implemented as ap_bus or ap_fifo
- **With a FIFO interface**
  - All reads & writes must be sequential (start from location 0 implied)
  - This functionality can only be implemented using ap_bus
- **Address generated by pointer arithmetic**
  - Can only exported externally using ap_bus
  - All other pointer interfaces are streaming type

Converting Malloc Pointers

- **Many design use Malloc and Pointer**
  - Malloc is not supported for synthesis
  - Must be converted to a resource of a defined size ➔ array
  - This can mean changing lots of code: *var ➔ var[ ]
- **The following workaround can ensure limited code impact**
  - Create a dummy variable
  - Set the pointer to the dummy variable

```
// Internal image buffers
#ifndef __SYNTHESIS__
  my_type *yuv = (my_type *)malloc(N*sizeof(my_type));
  my_type *scale = (my_type *)malloc(N*sizeof(my_type));
#else // Workaround malloc() calls w/o changing rest of code
  my_type _yuv[N];
  my_type _scale[N];
  my_type *yuv = &_yuv;
  my_type *scale = &_scale;
#endif
res = *yuv;
```

Code for non-synthesis flow (simulation)

Fixed size resource

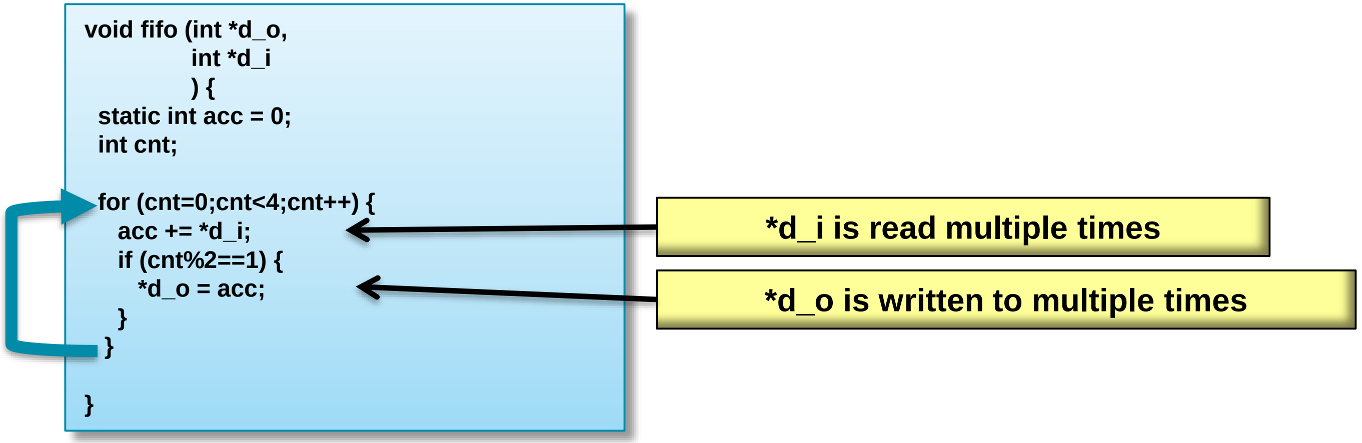
Point to new fixed sized resource

Keep pointer in code

Coding Considerations and IO

Coding and IO

- There is one major issue with IO and coding
  - Multi-access pointers
- A multi-access pointer is a pointer argument which is …
  - Read or written to multiple times



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

  for (cnt=0;cnt<4;cnt++) {
     acc += *d_i;
     if (cnt%2==1) {
        *d_o = acc;
     }
  }

}
```

*d_i is read multiple times

*d_o is written to multiple times

- This can result in incorrect hardware
- This cannot be verified using a test bench
- This can give incorrect results at RTL verification

Yes, you can use this coding style (if you REALLY wish)…. With the following workarounds

Multi-Access Pointers: Use Volatile!!!

- **Without Volatile**

```
void fifo (int *d_o,
                int *d_i) {
  static int acc = 0;
  int cnt;
  for (cnt=0;cnt<4;cnt++) {
     acc += *d_i;
     if (cnt%2==1) {
        *d_o = acc;
     }
   }
}
```

- **C compilers will optimize to 1 read and 1 write**

- **Vivado HLS will create a design with 1 read and 1 write**

To complicate matters, if the IF condition is complex, Vivado HLS may not be able to optimize the accesses into 1

- **With Volatile**

```
void fifo (volatile int *d_o,
               volatile int *d_i) {
  static int acc = 0;
  int cnt;
  for (cnt=0;cnt<4;cnt++) {
     acc += *d_i;
     if (cnt%2==1) {
        *d_o = acc;
     }
   }
}
```

- **C Compilers will not optimize the IO accesses**
  - Will assume the pointer value may change outside the scope of the function & leave the accesses

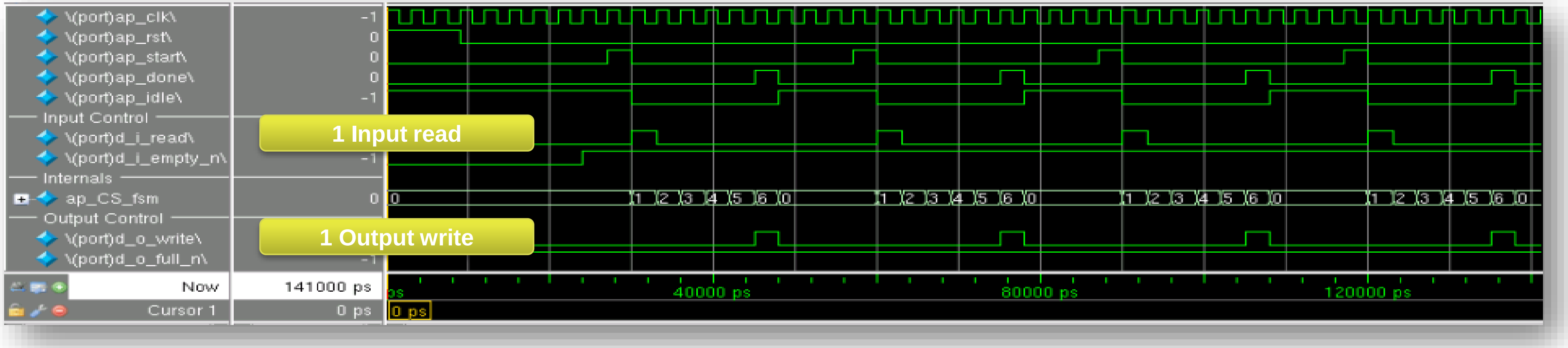
- **Vivado HLS will create a design with 4 reads and 2 write**

Multi-Access Pointers: the effect of volatile

C compilers will optimize multiple accesses on the same (non-volatile) pointer into a single access : Vivado HLS matches this behavior

> Easier to see with a simple example

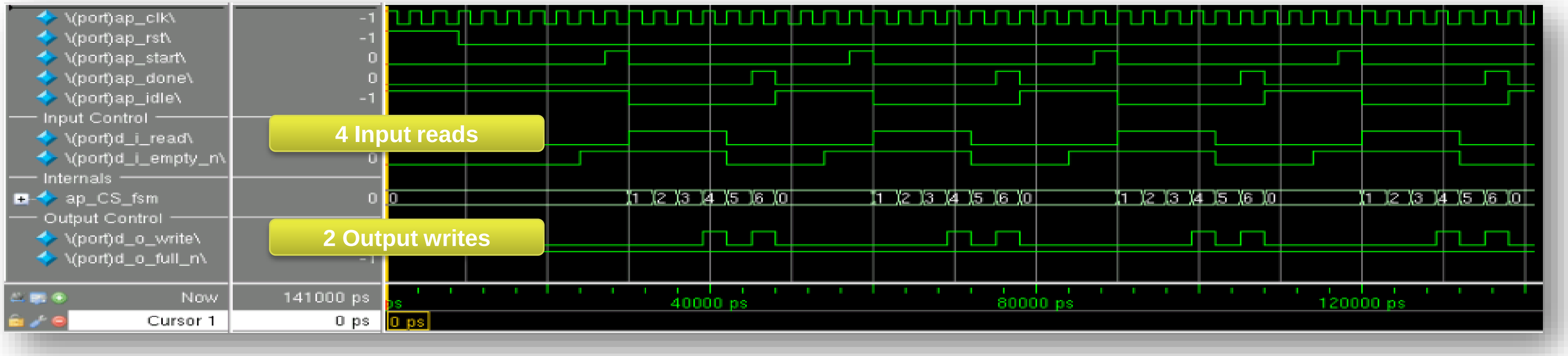
```
void fifo (
              int *d_o,
              int *d_i
) {
  static int acc = 0;
  int cnt;

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

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



1 Input read

1 Output write

```
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;
}
```



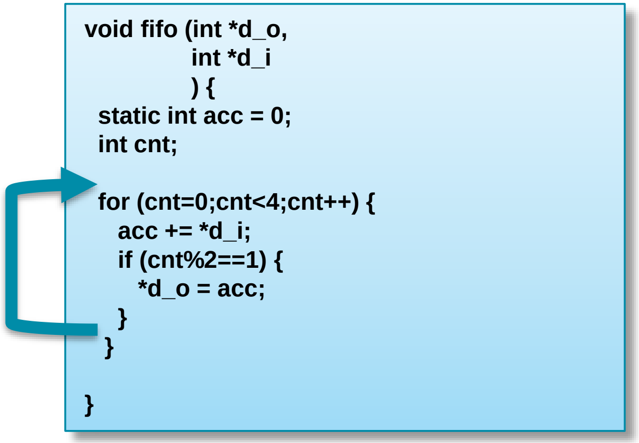
4 Input reads

2 Output writes

Multi-Access Pointers: Limited Visibility

> **The intermediate results are hard to use for verification**

  - Only the final pointer value is passed to the test bench



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

  for (cnt=0;cnt<4;cnt++) {
     acc += *d_i;
     if (cnt%2==1) {
        *d_o = acc;
     }
  }

}
```

- The final value of *d_o can be captured in the test bench and compared
- The intermediate values can only be seen by using printf : they cannot be automatically verified by the test bench

> **Code could be added to the DUT**

  - Use __SYNTHESIS__ to add unsynthesizable code for checking
  - Which will not be there in the RTL

Not the most ideal coding style for verification

  - HLS:STREAMS class can be used instead: discussed in next section

Multi-Access Pointers: Test bench Depth

> **Let’s look at an example**

- This code will access four samples using a pointer

```
#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;
  }
}
```

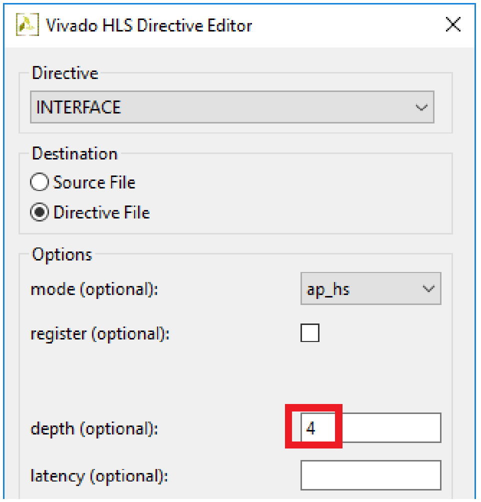
The C test bench may correctly provide 4 values

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

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

foo(d);

return 0;
}
```

> **Co-Simulation(RTL Verification) only sees a pointer on the interface**

- And will create a SystemC co-simulation wrapper to supply or store 1 sample
- Use the depth option to specify the actual depth
- Vivado HLS will issue a warning in case

@I [SIM-76] 'd' has a depth of ‘4'. Interface directive option '-depth' can be used to set to a different value. Incorrect depth may result in simulation mismtach.



Vivado HLS Directive Editor

Directive: INTERFACE

Destination
○ Source File
● Directive File

Options
mode (optional): ap_hs
register (optional): □
depth (optional): 4
latency (optional):

Streams

XILINX

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++ an as 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
- **Stream 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])

Libraries Support - FFT, FIR, Shift Registers and Linear Algebra

FFT and FIR IP in HLS

- **The Xilinx FFT and FIR IP are available in Vivado HLS**
  - C simulates with a bit-accurate model
  - Fully configurable within the C++ source code
    - Pre-defined C++ structs allow the IP to be configured & accessed
- **Supported only for C++**
  - Implemented with templates
- **High-Quality Implementation**
  - Same hardware as implemented by RTL versions of this IP
  - Functionality fully described in Xilinx Documentation
    - LogiCORE IP Fast Fourier Transform v9.0 (document PG109)
    - LogiCORE IP FIR Compiler v7.2 (document PG149)

FFT Function

- **Using the FFT**
  - Include the hls_fft.h library in the code
    - This defines the FFT and supporting structs and types
    - Allows hls::fft to be instantiated in your code
  - Use the STATIC_PARAM template parameter to parameterize the FFT
    - The STATIC_PARAM template parameter defines all static configuration values
    - The Library provides a pre-defined struct hls::ip_fft::params_t to perform this
  - Optionally modify the default parameters by creating a new user defined STATIC_PARAM struct based on the default

```
#include "hls_fft.h“

hls::fft<STATIC_PARAM> (                          // Static Parameterization Struct
  INPUT_DATA_ARRAY,                               // Input data fixed or float
  OUTPUT_DATA_ARRAY,                              // Output data fixed or float
  OUTPUT_STATUS,                                  // Output Status
  INPUT_RUN_TIME_CONFIGURATION);                  // Input Run Time Configuration
```

FFT Static Configuration

- **Default Configuration Pre-defined**
  - The hls_fft.h header file defines struct hls::ip_fft::params_t
  - This contains the default values for configuring the FFT
- **User defined struct**
  - Allows the FFT to be easily configured
  - Based on hls::ip_fft::params_t
  - User struct simply updates any specified value

```
 #include "hls_fft.h“

struct param1 : hls::ip_fft::params_t {
   static const unsigned ordering_opt = hls::ip_fft::natural_order;
   static const unsigned config_width = FFT_CONFIG_WIDTH;
   static const unsigned status_width = FFT_STATUS_WIDTH;
};

 // FFT IP
   hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);
```

Include Header File

Create struct based on hls::ip::fft:params_t

- **params_t includes the default values**
- **Struct *param1* overrides 3 default values in this example**
- **Struct *param1* is then used to configure the FFT**
- **Used as STATIC_PARAM in previous slide**

FFT Configuration Parameters

Parameter	Description
input_width	Data input port width
output_width	Data output port width
status_width	Output status port width
config_width	Input configuration port width
max_nfft	The size of the FFT data set is specified as 1 << max_nfft
has_nfft	Determines if the size of the FFT can be run time configurable
channels	Number of channels
arch_opt	The implementation architecture.
phase_factor_width	Configure the internal phase factor precision
ordering_opt	The output ordering mode
ovflo	Enable overflow mode
scaling_opt	Define the scaling options
rounding_opt	Define the rounding modes
mem_data	Specify using block or distributed RAM for data memory
mem_phase_factors	Specify using block or distributed RAM for phase factors memory
mem_reorder	Specify using block or distributed RAM for output reorder memory
stages_block_ram	Defines the number of block RAM stages used in the implementation
mem_hybrid	When block RAMs are specified for data, phase factor, or reorder buffer, mem_hybird specifies where or not to use a hybrid of block and distributed RAMs to reduce block RAM count in certain configurations
complex_mult_type	Defines the types of multiplier to use for complex multiplications
butterfly_type	Defines the implementation used for the FFT butterfly

These parameters are defined in param_t in hls_fft.h

FFT Configuration Defaults and Valid Values

Parameter	C Type	Default Value	Valid Values
input_width	unsigned	16	8-34
output_width	unsigned	16	input_width to (input_width + max_nfft + 1)
status_width	unsigned	8	Depends on FFT configuration
config_width	unsigned	16	Depends on FFT configuration
max_nfft	unsigned	10	3-16
has_nfft	bool	false	True, False
channels	unsigned	1	1-12
arch_opt	unsigned	pipelined_streaming_io	automatically_select, pipelined_streaming_io, radix_4_burst_io, radix_2_burst_io, radix_2_lite_burst_io
phase_factor_width	unsigned	16	8-34
ordering_opt	unsigned	bit_reversed_order	bit_reversed_order natural_order
ovflo	bool	true	false true
scaling_opt	unsigned	scaled	Scaled, unscaled, block_floating_point
rounding_opt	unsigned	truncation	truncation convergent_rounding
mem_data	unsigned	block_ram	block_ram distributed_ram
mem_phase_factors	unsigned	block_ram	block_ram distributed_ram
mem_reorder	unsigned	block_ram	block_ram distributed_ram
stages_block_ram	unsigned	(max_nfft < 10) ? 0 : (max_nfft - 9)	0-11
mem_hybrid	bool	false	false true
complex_mult_type	unsigned	use_mults_resources	use_luts use_mults_resources use_mults_performance
butterfly_type	unsigned	use_luts	use_luts use_xtremedsp_slices

FFT Function: Data

- **Input & Output Data**
  - Data Input and Output
    - Array arguments of type ap_fixed or float type
    - Two-dimensional arrays are used for multi-channel functionality
      - First dimension represents the channels; Second dimension the data for each channel
      - IP only supports multi-channels for ap_fixed (not float)

```
#include "hls_fft.h“

hls::fft<STATIC_PARAM> (                           // Static Parameterization Struct
 INPUT_DATA_ARRAY,                                 // Input data fixed or float
 OUTPUT_DATA_ARRAY,                                // Output data fixed or float
 OUTPUT_STATUS,                                    // Output Status
 INPUT_RUN_TIME_CONFIGURATION);                    // Input Run Time Configuration
```

FFT Function: Run Time Configuration

> Dynamic Configuration

- Parameter INPUT_RUN_TIME_CONFIGURATION can be changed at run time
- Access is made using the pre-defined struct hls::ip_fft::config_t<STATIC_PARAM>
  - Notice, this also uses the FFT parameterization struct

```
#include "hls_fft.h“

hls::fft<STATIC_PARAM> (                    // Static Parameterization Struct
  INPUT_DATA_ARRAY,                         // Input data fixed or float
  OUTPUT_DATA_ARRAY,                        // Output data fixed or float
  OUTPUT_STATUS,                            // Output Status
  INPUT_RUN_TIME_CONFIGURATION);            // Input Run Time Configuration
```

```
#include "hls_fft.h“

struct param1 : hls::ip_fft::params_t {
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
    static const unsigned config_width = FFT_CONFIG_WIDTH;
    static const unsigned status_width = FFT_STATUS_WIDTH;
};

typedef hls::ip_fft::config_t<param1> config_t;
config_t fft_config1;
```

```
// Set FFT length to 512 => log2(512) =>9
fft_config1-> setNfft(9);

// Forward FFT
fft_config1->setDir(0);
// Inverse FFT
fft_config1->setDir(1);

// Set FFT Scaling
fft_config1->setSch(0x2AB);

 // FFT IP
   hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);
```

FFT Function: Run Time Output Status

Dynamic Status Monitoring

- Parameter OUTPUT_STATUS can be read at run time
- Access is made using the pre-defined struct hls::ip_fft::status_t<STATIC_PARAM>
  - Notice, this also uses the FFT parameterization struct

```
#include "hls_fft.h“

hls::fft<STATIC_PARAM> (                         // Static Parameterization Struct
  INPUT_DATA_ARRAY,                              // Input data fixed or float
  OUTPUT_DATA_ARRAY,                             // Output data fixed or float
  OUTPUT_STATUS,                                 // Output Status
  INPUT_RUN_TIME_CONFIGURATION);                 // Input Run Time Configuration
```

```
#include "hls_fft.h“

struct param1 : hls::ip_fft::params_t {
    static const unsigned ordering_opt = hls::ip_fft::natural_order;
    static const unsigned config_width = FFT_CONFIG_WIDTH;
    static const unsigned status_width = FFT_STATUS_WIDTH;
};

typedef hls::ip_fft::status_t<param1> status_t;
status_t fft_status1;
```

```
 // FFT IP
    hls::fft<param1> (xn1, xk1, &fft_status1, &fft_config1);

// Check the overflow flag
    bool *ovflo = fft_status1->getOvflo();

// Obtain the block exponent
    unsigned int *blk_exp = fft_status1-> getBlkExp();
```

FIR Function

Using the FIR

- Include the hls_fir.h library in the code
  - This defines the FIR and supporting structs and types
  - Allows hls::FIR to be instantiated in your code
  - Unlike the FFT, the FIR is instantiated as a class and executed with the run method
- Create the STATIC_PARAM template parameter to configure the FIR
  - The STATIC_PARAM template parameter defines all static configuration values
  - The library provides a pre-defined struct hls::ip_fir::params_t to perform this
- There are no default values for the Coefficients
  - You Must Always create a user defined struct based on hls::ip_fir::params_t

```
#include "hls_fir.h“

// Create an instance of the FIR
static hls::FIR<STATIC_PARAM> fir1;        // Static parameterization

// Execute the FIR instance fir1
fir1.run(INPUT_DATA_ARRAY,                 // Input Data
         OUTPUT_DATA_ARRAY);               // Output Data
```

FIR Static Configuration

> **Default Configuration Pre-defined**
>> The hls_fir.h header file defines struct hls::ip_fir::params_t
>> This contains the default values for configuring the FIR

> **User defined struct must be created**
>> Allows the FIR to be easily configured
>> Based on hls::ip_fir::params_t
>> User struct simply updates any specified value

```
 #include "hls_fir.h“

struct param1 : hls::ip_fir::params_t {
   static const double coeff_vec[sg_fir_srrc_coeffs_len];
   static const unsigned num_coeffs = sg_fir_srrc_coeffs_len;
   static const unsigned input_width = INPUT_WIDTH;
   static const unsigned quantization = hls::ip_fir::quantize_only;};

// Execute FIR
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out);
```

Include Header File

Create struct based on hls::ip::fir:params_t

- **params_t includes the default values**
- **Struct *param1* overrides 3 default value in this example and specifies the coefficients**
    - **The coeff_vec has no default**
- **Struct *param1* is then used to configure the FIR**
- **Used as STATIC_PARAM in previous slide**

FIR Configuration Parameters

Parameter	Description
input_width	Data input port width
input_fractional_bits	Number of fractional bits on the input port
output_width	Data output port width
output_fractional_bits	Number of fractional bits on the output port
coeff_width	Bit-width of the coefficients
coeff_fractional_bits	Number of fractional bits in the coefficients
num_coeffs	Number of coefficients
coeff_sets	Number of coefficient sets
input_length	Number of samples in the input data
output_length	Number of samples in the output data
num_channels	Specify the number of channels of data to process
total_number_coeff	Total number of coefficients
coeff_vec[total_num_coeff]	The coefficient array
filter_type	The type implementation used for the filter
rate_change	Specifies integer or fractional rate changes
interp_rate	The interpolation rate
decim_rate	The decimation rate
zero_pack_factor	Number of zero coefficients used in interpolation
rate_specification	Specify the rate as frequency or period
hardware_oversampling_rate	Specify the rate of over-sampling
sample_period	The hardware oversample period
sample_frequency	The hardware oversample frequency
quantization	The quantitation method to be used
best_precision	Enable or disable the best precision
coeff_structure	The type of coefficient structure to be used
output_rounding_mode	Type of rounding used on the output
filter_arch	Selects a systolic or transposed architecture
optimization_goal	Specify a speed or area goal for optimization
inter_column_pipe_length	The pipeline length required between DSP columns
column_config	Specifies the number of DSP48 column
config_method	Specifies how the DSP48 columns are configured
coeff_padding	Number of zero padding added to the front of the filter

FIR Configuration Defaults and Valid Values

Parameter	C Type	Default Value	Valid Values
input_width	unsigned	16	No limitation
input_fractional_bits	unsigned	0	Limited by size of input_width
output_width	unsigned	24	No Limitation
output_fractional_bits	unsigned	0	Limited by size of output_width
coeff_width	unsigned	16	No Limitation
coeff_fractional_bits	unsigned	0	Limited by size of coeff_width
num_coeffs	bool	21	Full
coeff_sets	unsigned	1	1-1024
input_length	unsigned	21	No Limitation
output_length	unsigned	21	No Limitation
num_channels	unsigned	1	1-1024
total_number_coeff	unsigned	21	num_coeffs * coeff_sets
coeff_vec[total_num_coeff]	double array	None	Not applicable
filter_type	unsigned	single_rate	single_rate, interpolation, decimation, hibert, interpolated
rate_change	unsigned	integer	integer, fixed_fractional
interp_rate	unsigned	1	2-1024
decim_rate	unsigned	1	2-1024
zero_pack_factor	unsigned	1	2-8
rate_specification	unsigned	period	frequency,period
hardware_oversampling_rate	unsigned	1	No Limitation
sample_period	bool	1	No Limitation
sample_frequency	unsigned	0.001	No Limitation
quantization	unsigned	integer_coefficients	integer_coefficients , quantize_only, maximize_dynamic_range
best_precision	unsigned	false	false true
coeff_structure	unsigned	non_symetric	inferred , non_symetric, symmetric, negative_symmetric, half_band, hilbert
output_rounding_mode	unsigned	full_precision	full_precision, truncate_lsbs, non_symetric_rounding_down, non_symetric_rounding_up, symmetric_rounding_to_zero, symmetric_rounding_to_infinity, convergent_rounding_to_even, convergent_rounding_to_odd
filter_arch	unsigned	systolic_multiply_accumulate	systolic_multiply_accumulate, transpose_multiply_accumulate
optimization_goal	unsigned	area	area, speed
inter_column_pipe_length	unsigned	4	1-16
column_config	unsigned	1	Limited by number of DSP48s used
config_method	unsigned	single	single, by_channel
coeff_padding	bool	false	false true

FIR Function: Data

- **Input & Output Data**
  - Data Input and Output
    - Array arguments
- **Multi-channel functionality**
  - Supported through interleaving the data

```
 #include "hls_fir.h“

// Create an instance of the FIR
static hls::FIR<STATIC_PARAM> fir1;          // Static parameterization

// Execute the FIR instance fir1
fir1.run(INPUT_DATA_ARRAY,                   // Input Data
         OUTPUT_DATA_ARRAY);                 // Output Data
```

Interleave 2 channels into the FIR input

```
for (unsigned i = 0; i < LENGTH; ++i) {
     fir_input_data[2*i] = din_ch1[i];
     fir_input_data[2*i + 1] = din_ch2[i];
  }
```

De-Interleave 2 channels from the FIR output

```
for(unsigned i = 0; i < LENGTH; ++i) {
     dout_ch1[i] = fir_output_data[2*i];
     dout_ch2[i] = fir_output_data[2*i+1];
  }
```

FIR Function: Run Time Configuration

Dynamic Configuration

- Parameter INPUT_RUN_TIME_CONFIGURATION can be changed at run time
- In some modes a dynamic input can control how the coefficients are used
  - This is a standard 8-bit input

```
#include "hls_fir.h“

// Create an instance of the FIR
static hls::FIR<STATIC_PARAM> fir1;                    // Static parameterization

// Execute the FIR instance fir1
fir1.run(INPUT_DATA_ARRAY,                             // Input Data
         OUTPUT_DATA_ARRAY,                            // Output Data
         INPUT_RUN_TIME_CONFIGURATION);                // Run time input configuration
```

```
#include "hls_fir.h“

struct param1 : hls::ip_fir::params_t {
   static const double coeff_vec[sg_fir_srrc_coeffs_len];
   static const unsigned num_coeffs = sg_fir_srrc_coeffs_len;
   static const unsigned input_width = INPUT_WIDTH;
   static const unsigned quantization = hls::ip_fir::quantize_only;
};
```

```
// Define the configuration type
typedef ap_uint<8> config_t;

// Define the configuration variable
config_t fir_config = 8;

// Use the configuration in the FFT
static hls::FIR<param1> fir1;
fir1.run(fir_in, fir_out, &fir_config);
```

Using the FFT and FIR IP

- **FFT and FIR support pipelined implementations**
  - The functions themselves cannot be pipelined
  - They should be parameterized for pipelined operation
- **The data arguments are always arrays**
  - These will be implemented as AXI4 Streams in the RTL
    - By default, arrays are implemented as BRAM interfaces
- **Recommendation**
  - Use these IP in regions where dataflow optimization is used
  - This will auto-convert the input and output arrays into streaming arrays
- **Alternatively, a Requirement:**
  - The input and output arrays must be marked as streaming using the command set_directive_stream (pragma STREAM)

Shift Registers

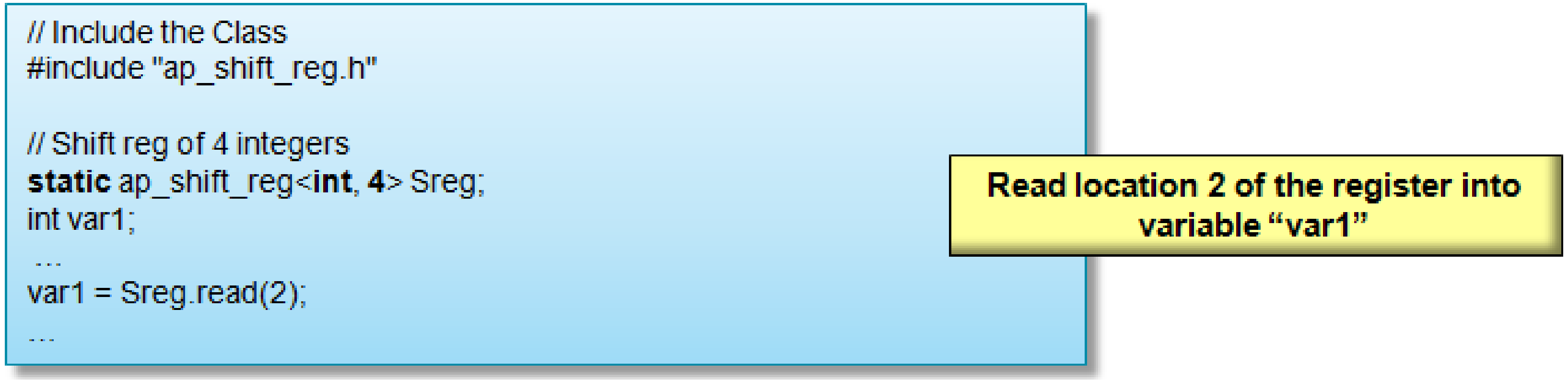
- **Many C algorithms sequentially shift data through arrays**
  - Add a new value to the start of the array,
  - Shift the existing data through array, and
  - Drop the oldest data value
- **Common way to implement a shift register from C into hardware**
  - Completely partition the array into individual elements, and allow the data dependencies between the elements in the RTL to imply a shift register
- **SRL architectural resource is most efficient to implement the desired shift registers**
  - When data is accessed in the middle of the shift register, logic synthesis cannot directly infer an SRL
  - Logic synthesis may implement the shift-resister in flip-flops, due to other factors
- **Use the `ap_shift_reg` class**

AP_SHIFT_REG Class

- **Class AP_SHIFT_REG infers an SRL**
  - Ensures that the behavior of an SRL component can be modeled in C
    - Addressable shift register
  - Ensures that an SRL is in fact used in the implementation
    - *Guarantees* that the high-performance Xilinx primitive is used
- **Using the AP_SHIFT_REG**
  - For use in C++ designs
  - Defined in the Vivado HLS tool header file *ap_shift_reg.h*
    - Must be included
- **SRL operations modeled in C**
  - Supported in the C code
  - Addressable read from the shift register
  - Addressable read from the shift register and shift
  - Addressable read from the shift register and shift if enabled

Using ap_shift_reg class

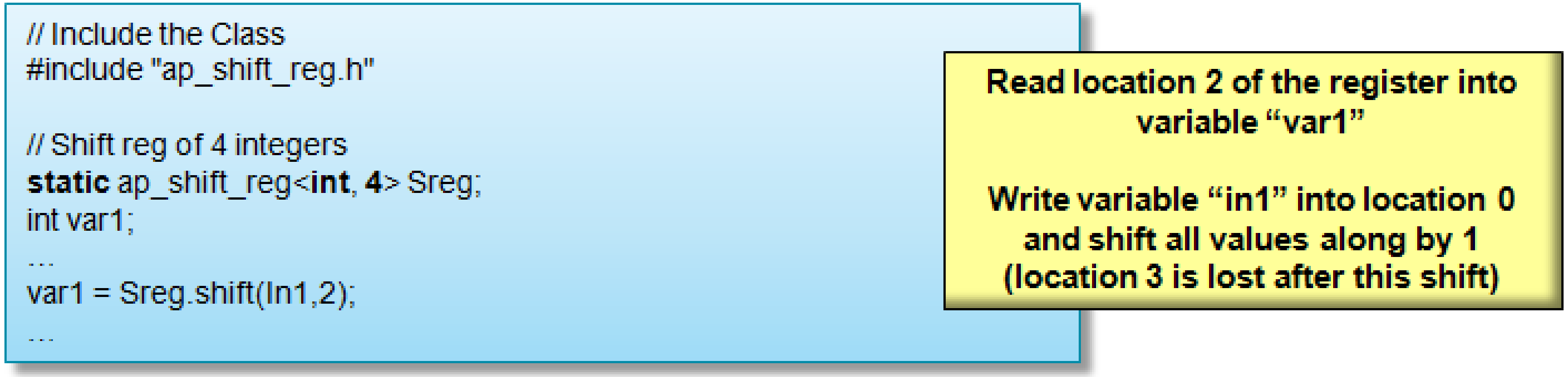
> **Read from the shift register**



```
// Include the Class
#include "ap_shift_reg.h"

// Shift reg of 4 integers
static ap_shift_reg<int, 4> Sreg;
int var1;
 …
var1 = Sreg.read(2);
…
```

Read location 2 of the register into variable “var1”

> **Read from and write to the shift register**



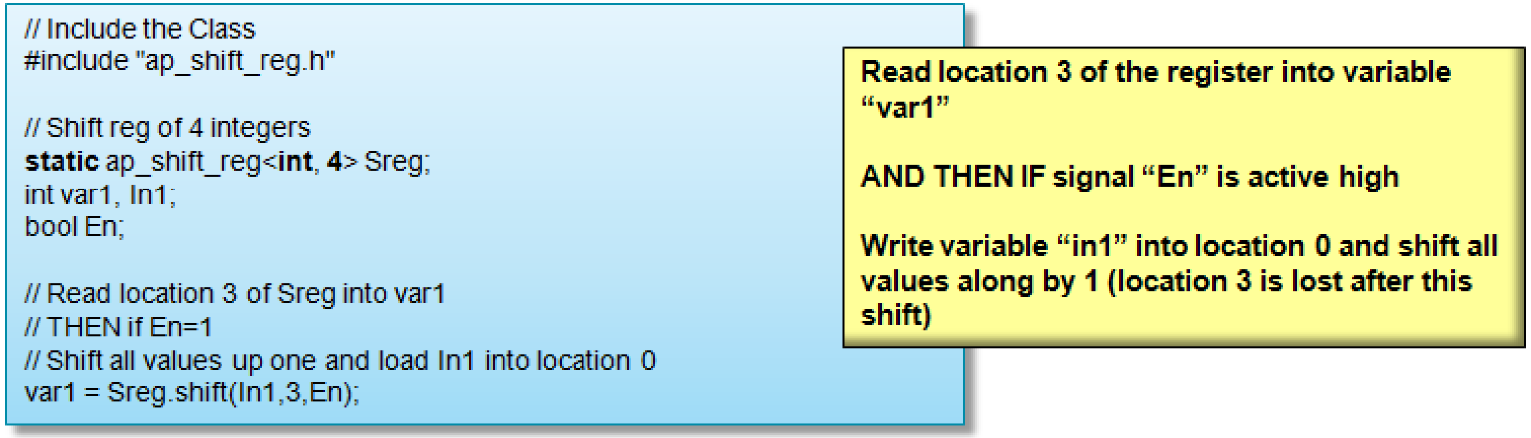
```
// Include the Class
#include "ap_shift_reg.h"

// Shift reg of 4 integers
static ap_shift_reg<int, 4> Sreg;
int var1;
…
var1 = Sreg.shift(In1,2);
…
```

Read location 2 of the register into variable “var1”

Write variable “in1” into location 0 and shift all values along by 1 (location 3 is lost after this shift)

Using ap_shift_reg class

> **Read from and optionally write to the shift register**



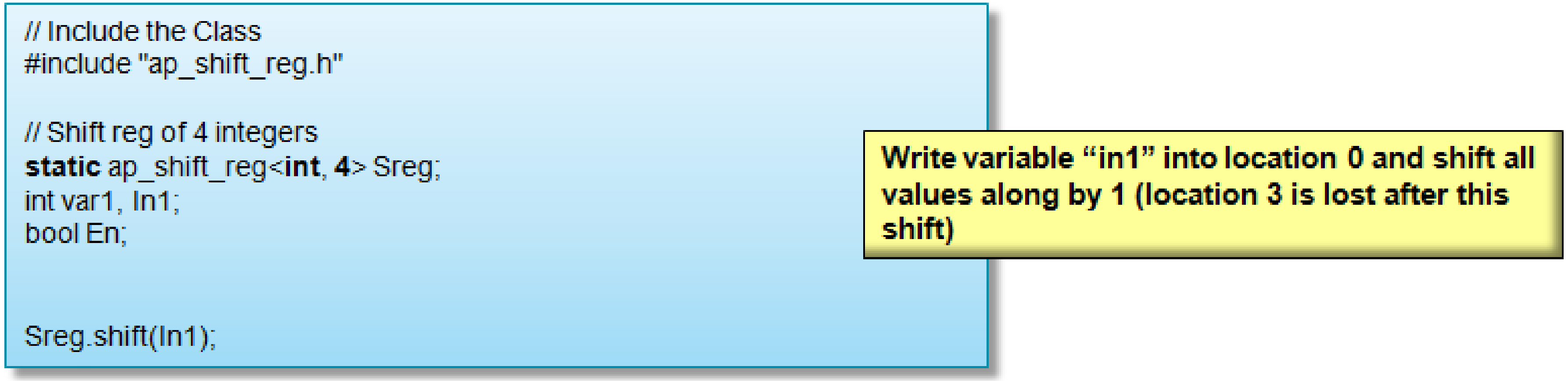
```
// Include the Class
#include "ap_shift_reg.h"

// Shift reg of 4 integers
static ap_shift_reg<int, 4> Sreg;
int var1, In1;
bool En;

// Read location 3 of Sreg into var1
// THEN if En=1
// Shift all values up one and load In1 into location 0
var1 = Sreg.shift(In1,3,En);
```

Read location 3 of the register into variable "var1"

AND THEN IF signal "En" is active high

Write variable "in1" into location 0 and shift all values along by 1 (location 3 is lost after this shift)

> **Shift only**



```
// Include the Class
#include "ap_shift_reg.h"

// Shift reg of 4 integers
static ap_shift_reg<int, 4> Sreg;
int var1, In1;
bool En;


Sreg.shift(In1);
```

Write variable "in1" into location 0 and shift all values along by 1 (location 3 is lost after this shift)

Linear Algebra Library

- **Provides a number of commonly used linear algebra functions**
  - All functions in this library use two-dimensional arrays to represent matrices

Functions	Data Types	Description
cholesky	float ap_fixed x_complex<float> x_complex<ap_fixed>	Computes the Cholesky decomposition of input matrix A, returning matrix L. Matrix L can be upper or lower triangular
cholesky_inverse	float ap_fixed x_complex<float> x_complex<ap_fixed>	Computes the inverse of symmetric positive definite input matrix A by the Cholesky decomposition method, producing matrix InverseA
matrix_multiply	float ap_fixed x_complex<float> x_complex<ap_fixed>	Computes the product of two matrices, returning a third matrix
qrf	float x_complex<float>	Computes the full QR factorization (QR decomposition) of input matrix A, producing orthogonal output matrix Q and upper-triangular matrix R
qr_inverse	float x_complex<float>	Computes the inverse of input matrix A by the QR factorization method, producing matrix InverseA
svd	float x_complex<float>	Computes the singular value decomposition of input matrix A, producing matrices U, S and V

Libraries Support - OpenCV

Video Libraries Supported in Vivado HLS

- **C video libraries**
  - Available within Vivado HLS tool header files
    - *hls_video.h* library
    - *hls_opencv.h* library
- **Enable migration of OpenCV designs for use with the Vivado HLS tool**
- **HLS video library is intended to replace many basic OpenCV functions**
  - Similar interfaces and algorithms to OpenCV
  - Focus on image processing functions implemented in FPGA fabric such as full HD video processing
  - Includes FPGA-specific optimizations
    - Fixed-point operations instead of floating point
    - On-chip line buffers and window buffers
  - Not necessarily bit-accurate
  - Libraries support standard AXI4 interfaces for easy system integration

Video Library Functions

- **C++ code contained in hls namespace: #include “hls_video.h”**
- **Similar interface; equivalent behavior with OpenCV**
  - OpenCV library: cvScale(src, dst, scale, shift);
  - HLS video library: hls::Scale<...>(src, dst, scale, shift);
- **Some constructor arguments have corresponding or replacement template parameters**
  - OpenCV library: cv::Mat mat(rows, cols, CV_8UC3);
  - HLS video library: hls::Mat<ROWS, COLS, HLS_8UC3> mat(rows, cols);
    - ROWS and COLS specify the maximum size of an image processed

Video Library Supported Functions

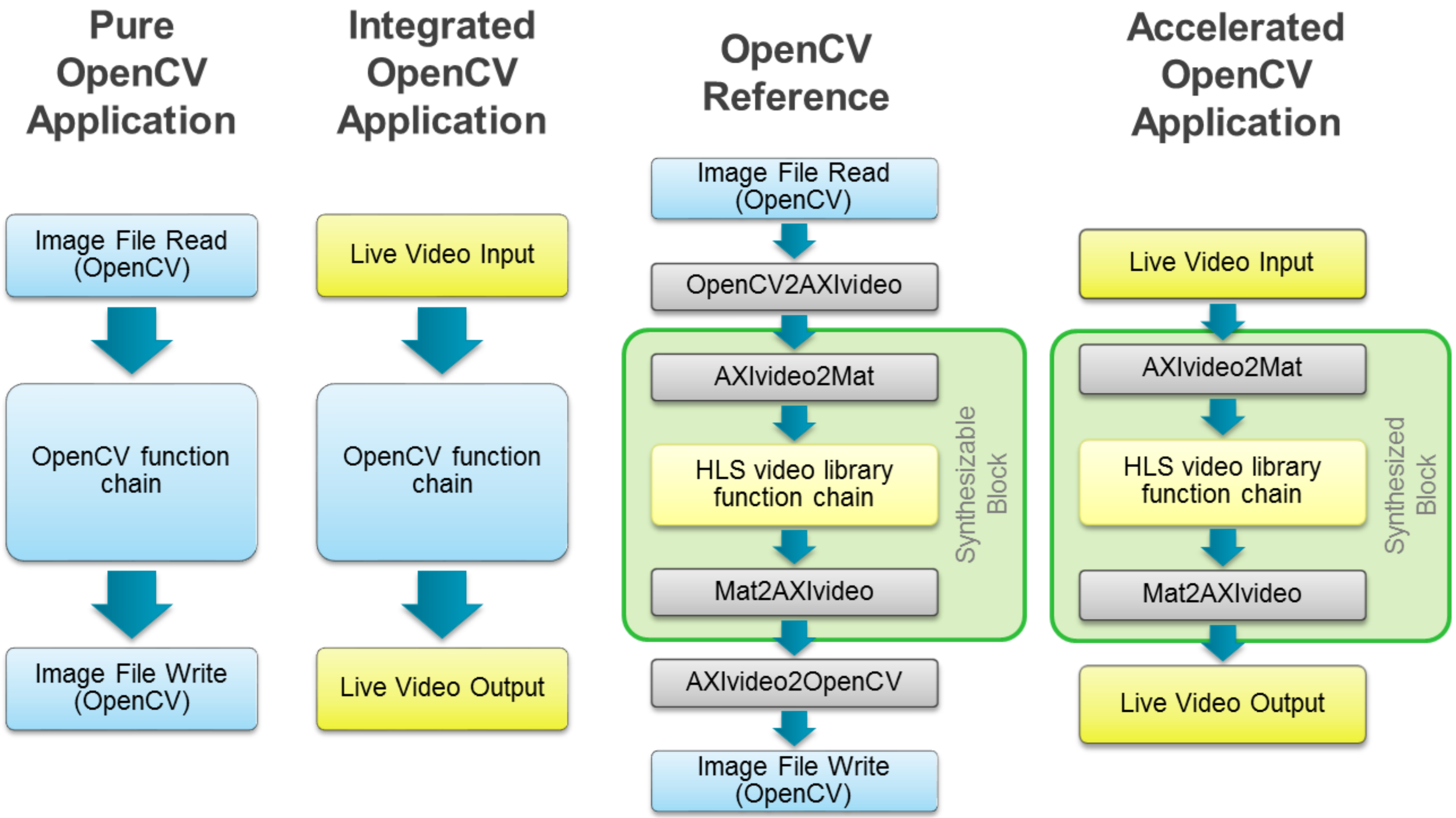
Video Data Modeling	
Linebuffer class	Window class

AXI4-Stream IO Functions	
AXIvideo2Mat	Mat2AXIvideo

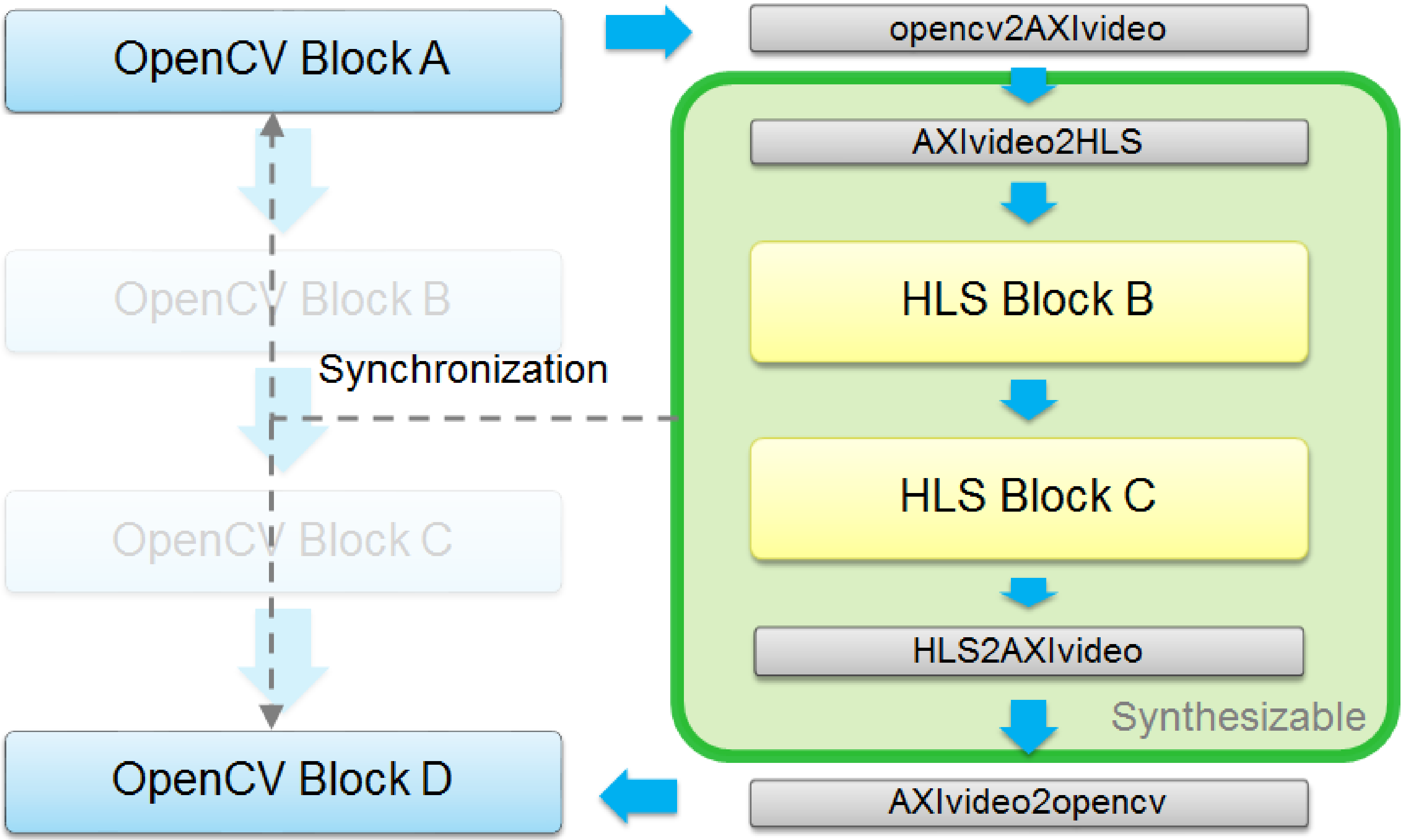
OpenCV Interface Functions			
cvMat2AXIvideo	AXIvideo2cvMat	cvMat2hlsMat	hlsMat2cvMat
IplImage2AXIvideo	AXIvideo2IplImage	IplImage2hlsMat	hlsMat2IplImage
CvMat2AXIvideo	AXIvideo2CvMat	CvMat2hlsMat	hlsMat2CvMat

Video Functions			
AbsDiff	Duplicate	MaxS	Remap
AddS	EqualizeHist	Mean	Resize
AddWeighted	Erode	Merge	Scale
And	FASTX	Min	Set
Avg	Filter2D	MinMaxLoc	Sobel
AvgSdv	GaussianBlur	MinS	Split
Cmp	Harris	Mul	SubRS
CmpS	HoughLines2	Not	SubS
CornerHarris	Integral	PaintMask	Sum
CvtColor	InitUndistortRectifyMap	Range	Threshold
Dilate	Max	Reduce	Zero

Using OpenCV in FPGA Designs

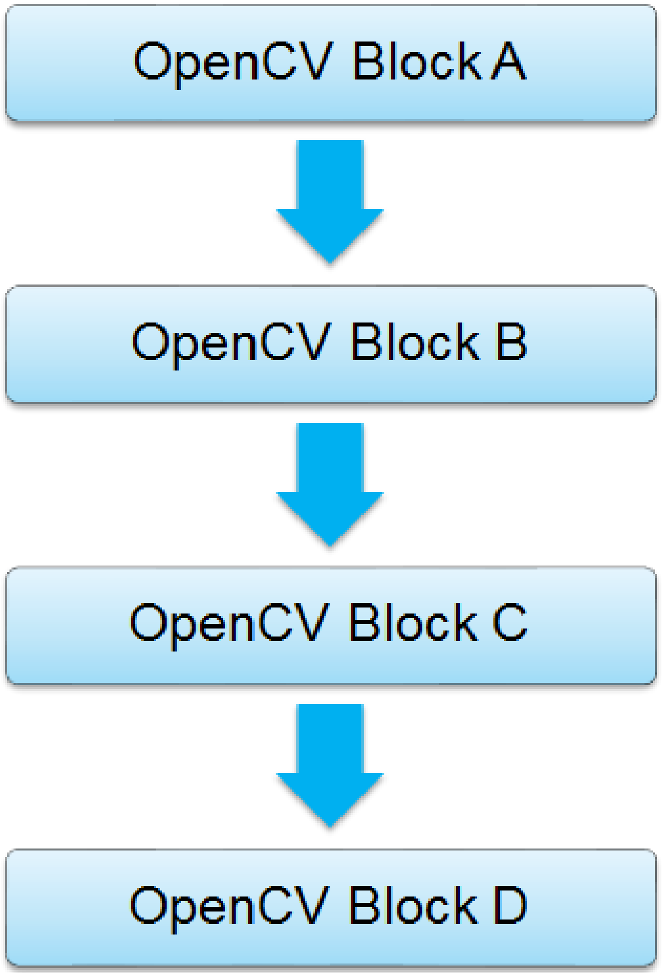


Partitioned OpenCV Application



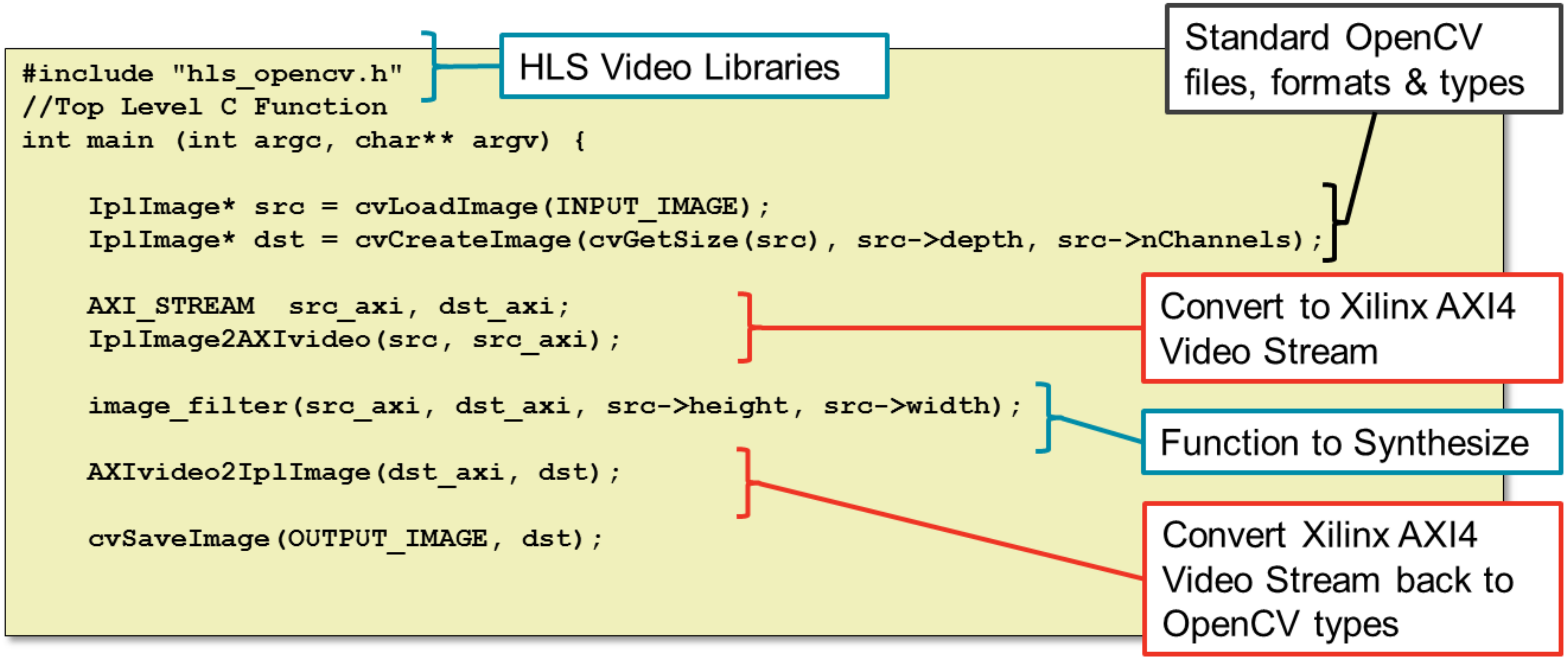
OpenCV Design Flow

- Develop OpenCV application on desktop
- Run OpenCV application on ARM® cores without modification
- Abstract FPGA portion using I/O functions
- Replace OpenCV function calls with synthesizable code
- Run HLS to generate FPGA accelerator
- Replace call to synthesizable code with call to
  FPGA accelerator



C Testbench: Interface Library Example

- **Interface libraries convert to/from OpenCV image to HLS type**
  - HLS MAT format: synthesizable and AXI4 Stream support



```
#include "hls_opencv.h"
//Top Level C Function
int main (int argc, char** argv) {

    IplImage* src = cvLoadImage(INPUT_IMAGE);
    IplImage* dst = cvCreateImage(cvGetSize(src), src->depth, src->nChannels);

    AXI_STREAM  src_axi, dst_axi;
    IplImage2AXIvideo(src, src_axi);

    image_filter(src_axi, dst_axi, src->height, src->width);

    AXIvideo2IplImage(dst_axi, dst);

    cvSaveImage(OUTPUT_IMAGE, dst);
```

HLS Video Libraries

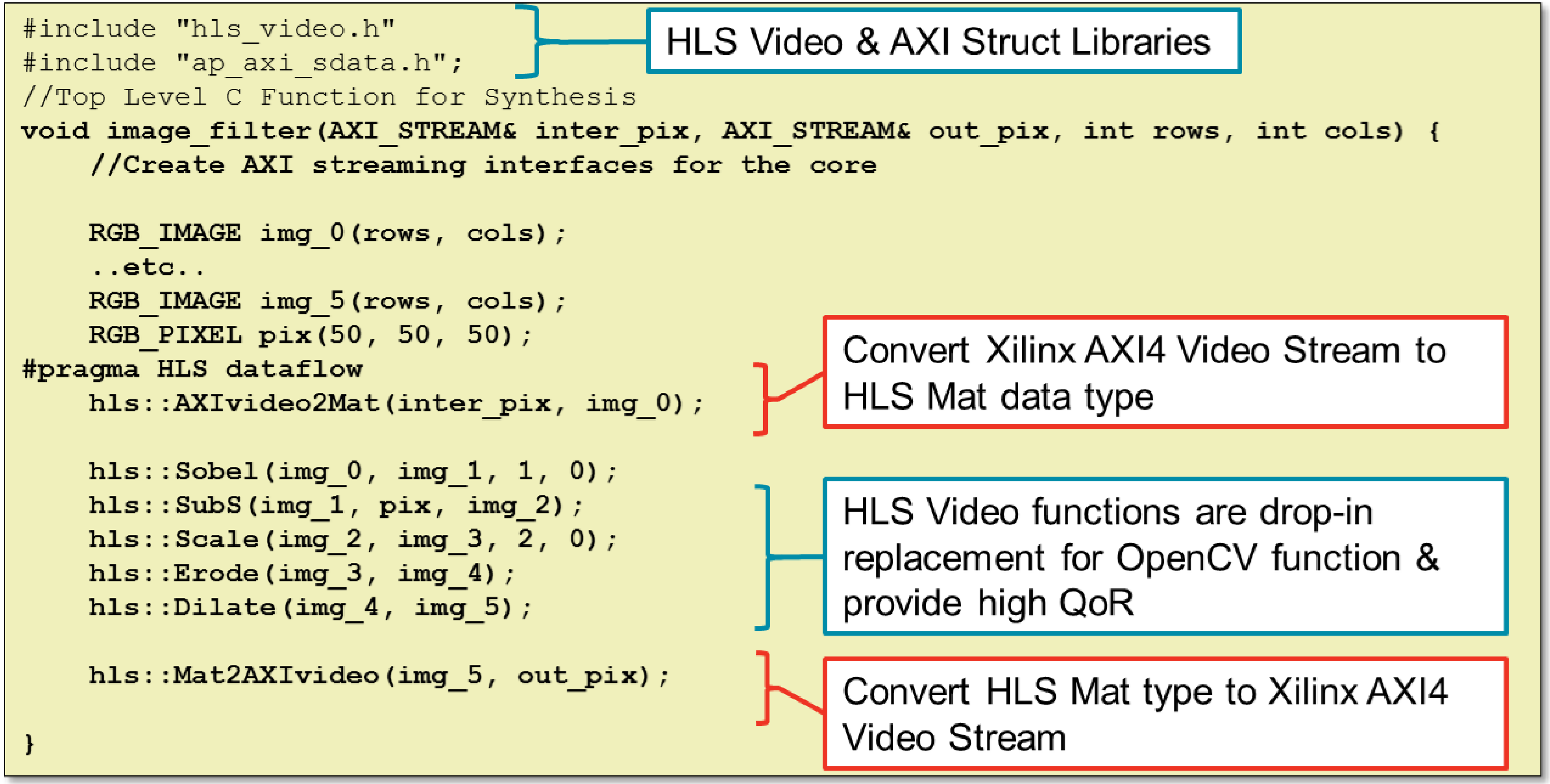
Standard OpenCV files, formats & types

Convert to Xilinx AXI4 Video Stream

Function to Synthesize

Convert Xilinx AXI4 Video Stream back to OpenCV types

Synthesizable C Function Example



```
#include "hls_video.h"
#include "ap_axi_sdata.h";
//Top Level C Function for Synthesis
void image_filter(AXI_STREAM& inter_pix, AXI_STREAM& out_pix, int rows, int cols) {
    //Create AXI streaming interfaces for the core

    RGB_IMAGE img_0(rows, cols);
    ..etc..
    RGB_IMAGE img_5(rows, cols);
    RGB_PIXEL pix(50, 50, 50);
#pragma HLS dataflow
    hls::AXIvideo2Mat(inter_pix, img_0);

    hls::Sobel(img_0, img_1, 1, 0);
    hls::SubS(img_1, pix, img_2);
    hls::Scale(img_2, img_3, 2, 0);
    hls::Erode(img_3, img_4);
    hls::Dilate(img_4, img_5);

    hls::Mat2AXIvideo(img_5, out_pix);

}
```

HLS Video & AXI Struct Libraries

Convert Xilinx AXI4 Video Stream to HLS Mat data type

HLS Video functions are drop-in replacement for OpenCV function & provide high QoR

Convert HLS Mat type to Xilinx AXI4 Video Stream

Summary

XILINX®

Summary

- **Vivado HLS supports C, C++, and SystemC**
  - Vast majority of constructs are synthesizable provided they are statically defined at compile time
- **Some of the unsupported constructs are**
  - System calls and function pointer
  - Forward declared data type
  - Recursive type functions
  - Dynamic memory allocation
  - Dynamic virtual calls
  - Pointer reinterpretation
- **Multi-access pointer is a pointer which is read and written to multiple times**
  - Must be marked as volatile or reads and writes will be optimized away
  - Cannot be rigorously validated with C simulation
  - Require a depth specification for RTL simulation

Summary

- **When struct is passed as a pointer the data are accessed as pass-by-reference**
- **When struct is not passed a pointer the data are accessed as by value**
- **When pointer arithmetic is performed, argument may be implemented as ap_bus or ap_fifo**
- **Stream can be used instead of multiple-access pointer**
- **Vivado HLS provides support for FFT, FIR, SRL, and Linear Algebra**
- **Stream and shift registers classes may model and implement hardware better**
- **HLS video libraries**
  - Drop-in replacement for OpenCV and provide high QoR