

C-based Design: High-Level Synthesis with the Vivado HLS Tool

dsp-hls-2016.1-wkbp-rev1

C-based Design: High-Level Synthesis with the Vivado HLS Tool 2016.1



© Copyright 2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, UltraScale, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. ARM is a registered trademark of ARM in the EU and other countries. All other trademarks are the property of their respective owners.

DISCLAIMER

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of the Limited Warranties which can be viewed at <http://www.xilinx.com/warranty.htm>; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in Critical Applications: <http://www.xilinx.com/warranty.htm#critapps>. All other trademarks are the property of their respective owners.

Table of Contents

Course Agenda	1-1
Build Better Systems Faster – Impact of this Class	1-1
Build Better Systems Faster – What Will You Learn? (1)	1-2
Build Better Systems Faster – What Will You Learn? (2)	1-2
Course Objectives	1-3
Introduction to High-Level Synthesis and the Vivado HLS Tool	2-1
Objectives	2-1
Introduction to High-Level Synthesis	2-2
Basics of the Vivado HLS Tool	2-9
Design Exploration	2-17
Language Support	2-22
Validation Flow	2-25
Validation Flow	2-28
Summary	2-31
Using the Vivado HLS Tool	3-1
Objectives	3-1
Invoking the Vivado HLS Tool	3-2
Project Creation in the Vivado HLS Tool	3-6
C Validation to IP Creation	3-13
Adding Directives	3-27
Vivado HLS Tool Directory Structure	3-32
Summary	3-34
I/O Interfaces	4-1
Objectives	4-1

HLS UltraFast Design Methodology	4-2
Step 1: Initial Optimizations – Directives	4-3
Overview of Vivado HLS Tool I/O Ports, Protocols, and Options	4-4
Block-Level I/O Protocols	4-13
Port-Level I/O Protocols	4-18
Port-Level I/O Protocols: AXI Interfaces	4-22
Port-Level I/O Protocols: AXI Interfaces	4-35
Port-Level I/O Protocols: Memory Interfaces	4-36
DATA_PACK Directive	4-46
Summary	4-49
Pipelining for Performance	5-1
Objectives	5-1
HLS UltraFast Design Methodology	5-2
Step 2: Pipeline for Performance	5-3
Pipelining	5-4
Dataflow Optimization	5-9
Summary	5-18
Optimizing Structures for Performance	6-1
Objectives	6-1
HLS UltraFast Design Methodology	6-2
Step 3: Optimize Structures for Performance (1)	6-3
Step 3: Optimize Structures for Performance (2)	6-3
Arrays in HLS	6-4
Array Optimizations	6-6
Summary	6-17
Reducing Latency	7-1
Objectives	7-1

HLS UltraFast Design Methodology	7-2
Step 4: Reduce Latency	7-2
Improving Latency	7-3
Loops: Impact on Latency	7-7
Summary	7-15
Improving Area	8-1
Objectives	8-1
HLS UltraFast Design Methodology	8-2
Step 5: Improve Area (1)	8-3
Step 5: Improve Area (2)	8-4
Improving Area	8-4
Controlling the Resources Used	8-5
Controlling the Structure of the Design	8-8
Importance of Arbitrary Precision Types	8-15
Summary	8-16
Introduction to the HLx Design Flow	9-1
Objectives	9-1
Productivity Improvement Opportunity	9-2
RTL vs C-Based Design	9-2
Traditional RTL vs Vivado HLx Design Flow	9-3
Vivado HLS Tool Flow	9-5
User-Preferred System Integration Environment	9-6
HLS vs. SDSoc Development Environment Flow	10-1
Objectives	10-1
Vivado HLS Tool Flow	10-2
User-Preferred System Integration Environment	10-3
Scope of the SDSoc Development Environment	10-4

Benefits of Using the SDSoc Development Environment	10-5
Development Flow Without the SDSoc Tool	10-6
Development Flow Using the SDSoc Tool	10-7
Process and Results of a Typical SDSoc Development Environment Process	10-9
What Users Need to Know to Be Highly Successful	10-10
Vivado HLS Tool: C Code	11-1
Objectives	11-1
C Validation Flow	11-2
C Language Support	11-10
Data Types and Bit Accuracy	11-20
C apint Types	11-22
C++ ap_int Types	11-23
C++ ap_fixed Types	11-27
SystemC Types	11-36
Floating-Point Types	11-38
Pointers	11-43
Hardware Modeling	11-52
OpenCV Libraries	11-57
Summary	11-61
Appendix: I/O Interfaces	12-1
Vivado HLS Tool I/O Ports, Protocols, and Options	12-1
Port-Level I/O Protocols: Wire Handshakes	12-12
Coding Issues and I/O	12-17
HLS UltraFast Design Methodology	13-1
HLS UltraFast Design Methodology	13-1
Summary	13-31

Appendix: SystemC Synthesis	14-1
SystemC Synthesis Coverage	14-1
SystemC Constructs	14-3
TLM Synthesis	14-9
Multi-Clock Domains	14-14
Vivado HLS Tool SystemC Extensions	14-15
Tutorial Example	14-17
Coding Styles for SystemC (Complements to C Coding Style)	14-19
Differences Between Vivado HLS Tool and SystemC Types	14-25
Appendix	15-1
Additional Xilinx Courses	15-1

Course Agenda

Slide 1-1:

2016.1

This module covers the agenda for the course. 10 minutes.

Slide 1-2:

Build Better Systems Faster – Impact of this Class



Accelerated Design Productivity

- Vivado® High-Level Synthesis (HLS) tool; generates RTL from C-based design sources

Increased System Performance

BOM Cost Reduction

Total Power Reduction

- Improve throughput, latency, resource utilization of the design using optimized design techniques

Programmable Systems Integration

- Generate IP for use with an RTL-based design in the Vivado Design Suite, a block-based design with the Vivado IP integrator, a model-based design in the System Generator for DSP or as a pcore for use in the Xilinx Platform Studio

ARTIX⁷

KINTEX⁷

VIRTEX⁷

ZYNQ

VIVADO

Slide 1-3:

Build Better Systems Faster – What Will You Learn? (1)



- Accelerated design productivity
 - Design reuse
 - More turns per day
 - Quick time to market
 - All modules + all labs
- Increased system performance
 - Utilize directives to improve performance
 - "Pipelining for Performance" module + demo + lab
 - "Optimizing Structures for Performance" module + demo + lab
 - "Improving Area" module + lab
 - Use the Video Library to implement OpenCV functions on FPGA for high performance
 - "Vivado HLS Tool: C Code" module

Slide 1-4:

Build Better Systems Faster – What Will You Learn? (2)



- BOM cost reduction and total power reduction
 - Optimize resource utilization by applying proper directives
 - "Improving Area" module + lab
 - Modeling hardware-efficient hardware from the C design
 - "Vivado HLS Tool: C Code" module + lab
- Programmable system integration
 - Generate IP for use with an RTL-based design in the Vivado Design Suite, a block-based design with the Vivado IP integrator, a model-based design in the System Generator for DSP, or as a pcore for use in the Xilinx Platform Studio
 - "I/O Interfaces" module + lab
 - AXI4-Streaming Interfaces demo
 - "Introduction to the HLx Design Flow" + demo + lab

Slide 1-5:

Course Objectives



After completing this course, you will be able to:

- Enhance productivity by using the Vivado HLS tool
- Describe the high-level synthesis flow
- Use the Vivado HLS tool for a first project
- Identify the importance of the testbench
- Use directives to improve performance and area and select RTL interfaces
- Identify common coding pitfalls as well as methods for improving code for RTL/hardware
- Perform system-level integration of blocks generated by the Vivado HLS tool
- Describe how to use OpenCV functions in the Vivado HLS tool

Slide 1-6:

Prerequisites



- Basic knowledge of C, C++, or SystemC
- or
- *High-Level Synthesis for Hardware Engineers* course
- Basic digital design concepts
- or
- *High-Level Synthesis for Software Engineers* course

Slide 1-7:

Day One Agenda



- Introduction to High-Level Synthesis and the Vivado HLS Tool (60 min.)
- Using the Vivado HLS Tool (60 min.)
- Demo: Vivado HLS Tool Overview (20 min.)
- Lab 1: Introduction to the Vivado HLS Tool Flow (45 min.)
- Lab 2: Introduction to the Vivado HLS Tool CLI Flow (30 min.)
- I/O Interfaces (75 min.)
- Demo: AXI4-Stream Interfaces (20 min.)
- Lab 3: Interface Synthesis (45 min.)
- Pipelining for Performance (45 min.)
- Demo: Pipelining for Performance (20 min.)
- Lab 4: Improving Performance (45 min.)

Slide 1-8:

Day Two Agenda



- Optimizing Structures for Performance (30 min.)
- Demo: Using Vivado HLS IP with SysGen (15 min.)
- Demo: Handling Memories (20 min.)
- Lab 7: HLx Flow - System Integration (75 min.)
- Lab 5: Implementing Arrays as RTL Interfaces (45 min.)
- HLS vs. SDSoc Development Environment Flow (15 min.)
- Reducing Latency (45 min.)
- Demo: SDSoc Tool Overview (20 min.)
- Improving Area (30 min.)
- Vivado HLS Tool: C Code (75 min.)
- Lab 6: Improving Area and Resource Utilization (60 min.)
- Introduction to the HLx Design Flow (10 min.)

Slide 1-9:

Where Can I Learn More?



- *High-Level Synthesis Design Hub in DocNav*
 - One stop point for all HLS collateral (user guide, tutorials, QuickTake videos, Application Notes, and support resources)

High-Level Synthesis (C based)		
V2014.3 - Published 2014-09-10		
<input checked="" type="checkbox"/> Getting Started	Published	
Introduction		
Vivado High Level Synthesis		
Getting Started with Vivado High-Level Synthesis		
Vivado Design Suite Tutorial: High-Level Synthesis		
Introduction to FPGA Design Using High-Level Synthesis		
Key Concepts	Published	
Analyzing your Vivado HLS Design for Performance Improvements		
Packaging Vivado HLS IP for use from Vivado IP Catalog		
Specifying AXI4 Interfaces for your Vivado HLS Design		
Verifying your Vivado HLS Design		
Properly Defining Interfaces in High-Level Synthesis		
Recommended Coding Styles		
Design Optimization		
<input checked="" type="checkbox"/> Additional Learning Materials		
User Guides	Design File(s)	Published
Vivado Design Suite User Guide: High-Level Synthesis (HLS)		
Videos	Design File(s)	Published
Using the Vivado High-Level Synthesis (HLS) Tcl Interface		
Floating Point Design with Vivado High-Level Synthesis (HLS)		

Slide 1-10:



Appendix

- Note that this course also includes the following appendixes
 - Appendix: "HLS UltraFast Design Methodology"
 - Appendix: "I/O Interfaces"
 - Appendix: "SystemC Synthesis"

Slide 1-11:

Latest Product Information

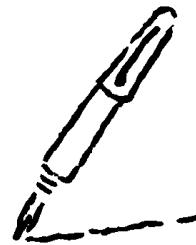


Please visit the following resources for the most current information on the Xilinx devices described in this course.

- User design information can be found in user guides
- IP core information can be found in product guides
- Characteristics, such as timing, performance, etc., are listed in data sheets
- For design and software issues or bugs, see www.xilinx.com
 - > Support
- DocNav provides easy access to these documents



Capture Your Notes Here



Introduction to High-Level Synthesis and the Vivado HLS Tool

Slide 2-1:

2016.1



In this module, you will explore the default behavior and optimizations of the Vivado® High-Level Synthesis (HLS) tool: how it implements functions, loops, arrays, and I/O ports. Later modules will show how a design can be modified using optimization directives. 60 minutes.

Slide 2-2:

Objectives



After completing this module, you will be able to:

- Describe the need for high-level synthesis
- Identify the steps to extract RTL from C using Vivado HLS
- Describe the basic terminology used in HLS
- Describe how directives can impact the performance and area objectives of a C design
- Perform C language support for Vivado HLS

Slide 2-3:

Introduction to High-Level Synthesis



■ Introduction to High-Level Synthesis

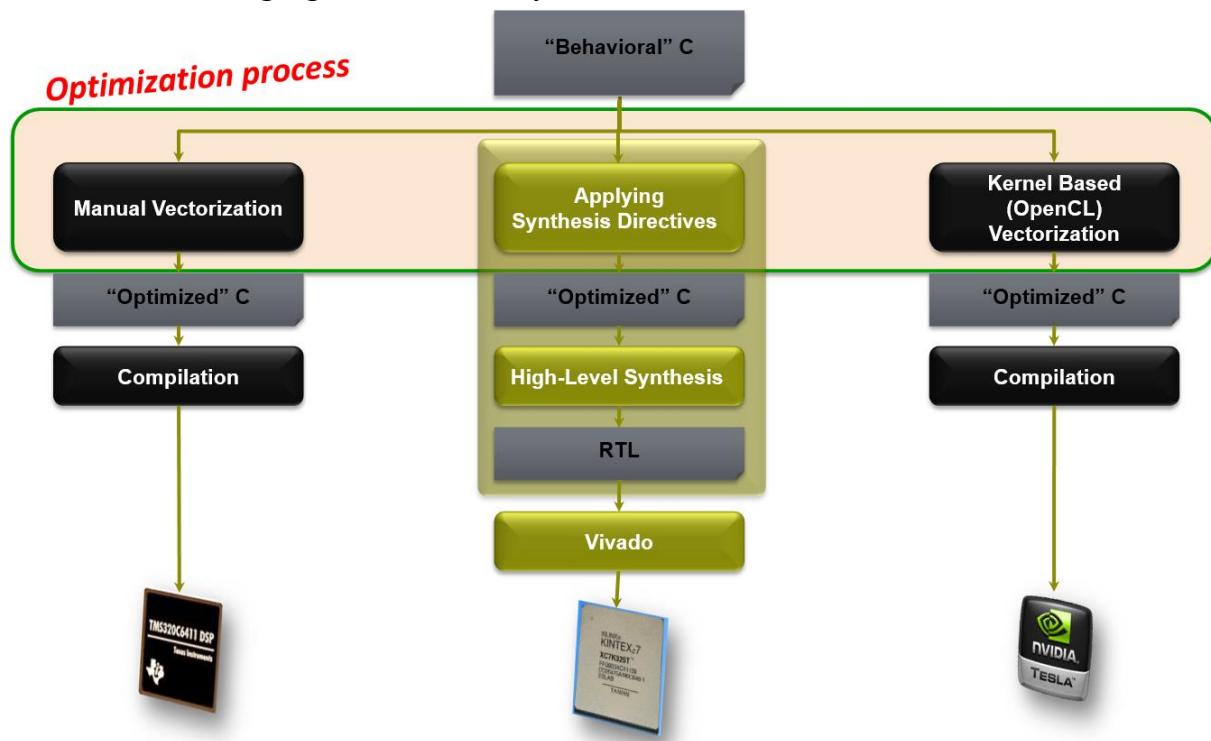
- Basics of the Vivado HLS Tool
- Design Exploration
- Language Support
- Validation Flow
- HLS UltraFast Design Methodology
- Summary

Slide 2-4:

High-Level Synthesis



- C-based design flow is optimized for hardware targets
 - Makes design goals more easily achieved





Algorithmic-based approaches are popular due to accelerated design time and time-to-market pressures. Larger designs pose challenges in the design and verification of hardware.

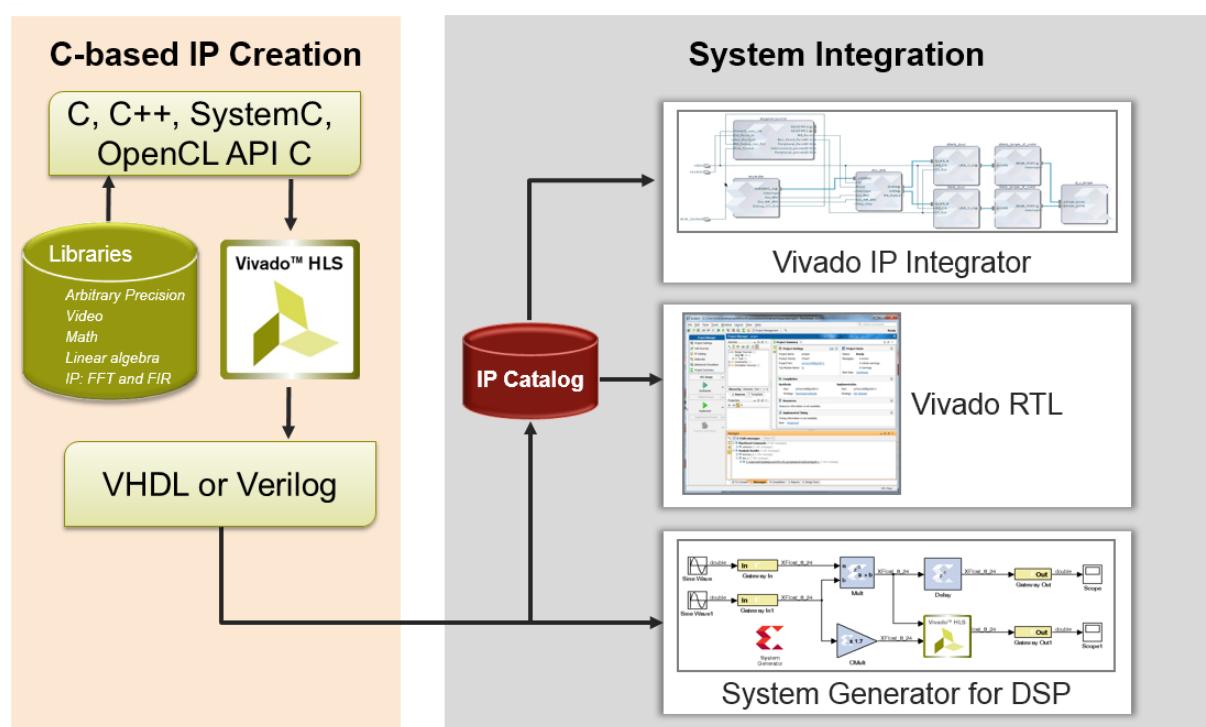
The industry trend is moving toward hardware acceleration to enhance performance and productivity. CPU-intensive tasks are now offloaded to a hardware accelerator. Hardware accelerators require a lot of time to understand and design.

The Vivado HLS tool converts algorithmic descriptions written in C-based design flow into hardware description (RTL). This elevates the abstraction level from RTL to algorithms.

High-level synthesis is essential for maintaining design productivity in large designs.

Slide 2-5:

User-Preferred System Integration Environment



Here is a more in-depth, or system view, of the Vivado HLS tool. As you can see, the input to the Vivado HLS tool is C, C++, SystemC, or OpenCL API C.

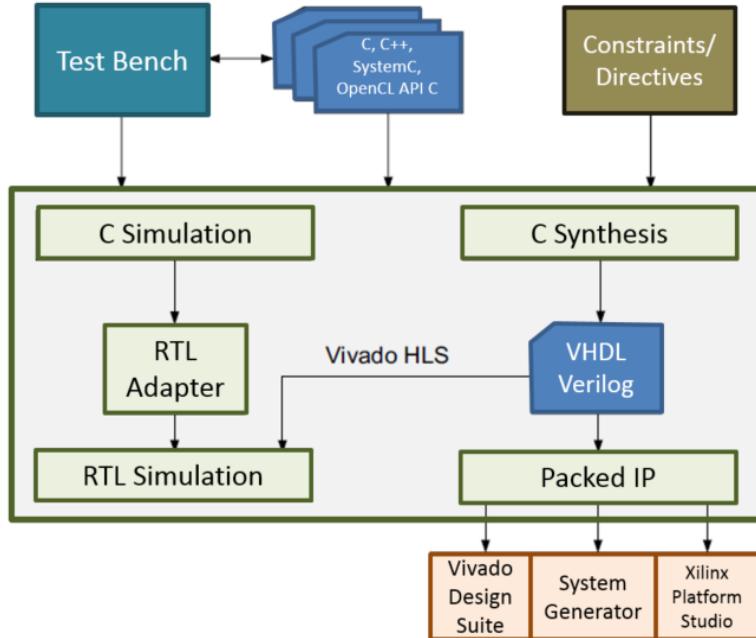
In this diagram, you will also notice that the Vivado HLS C and C++ are including C Libraries and a *math.h*, some video libraries, or some DSP libraries available as part of the Vivado HLS tool library. The Vivado HLS tool uses a number of C libraries to improve the performance of the designs.

When the output of the Vivado HLS tool is available, it can be exported into the IP catalog and throughout the rest of the Vivado Design Suite. It can be imported as an RTL block into Vivado RTL, used in the Vivado IP integrator, and be used as a single block in a system generator for DSP.

Slide 2-6:

Vivado HLS Tool Flow

-  Start with C, C++, SystemC, or OpenCL C
 - Design source and testbench code
- C to RTL synthesis
 - Schedule and map resources
 - Generate generic RTL code
 - Optimize using synthesis directives
- RTL validation
 - Uses the original C testbench
 - RTL simulation (XSim/ISim/ModelSim)
- Export to
 - IP catalog
 - System Generator
 - PCore



 The following are the *inputs* to the Vivado HLS tool:

- **C function written in C, C++, SystemC, or an OpenCL API C kernel:** This is the primary input to the Vivado HLS tool. The function can contain a hierarchy of sub-functions.
- **Constraints:** Constraints are required and include the clock period, clock uncertainty, and FPGA target. The clock uncertainty defaults to 12.5% of the clock period if not specified.
- **Directives:** Directives are optional and direct the synthesis process to implement a specific behavior or optimization.

- **C testbench and any associated files:** The Vivado HLS tool uses the C testbench to simulate the C function prior to synthesis and to verify the RTL output using C/RTL co-simulation.

The following are the *outputs* from the Vivado HLS tool:

- **RTL implementation files in hardware description language (HDL) formats:**

This is the primary output from the Vivado HLS tool. Using Vivado synthesis, you can synthesize the RTL into a gate-level implementation and an FPGA bitstream file. The RTL is available in the following-industry standard formats:

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

The Vivado HLS tool packages the implementation files as an IP block for use with other tools in the Xilinx design flow. Using logic synthesis, you can synthesize the packaged IP into an FPGA bitstream.

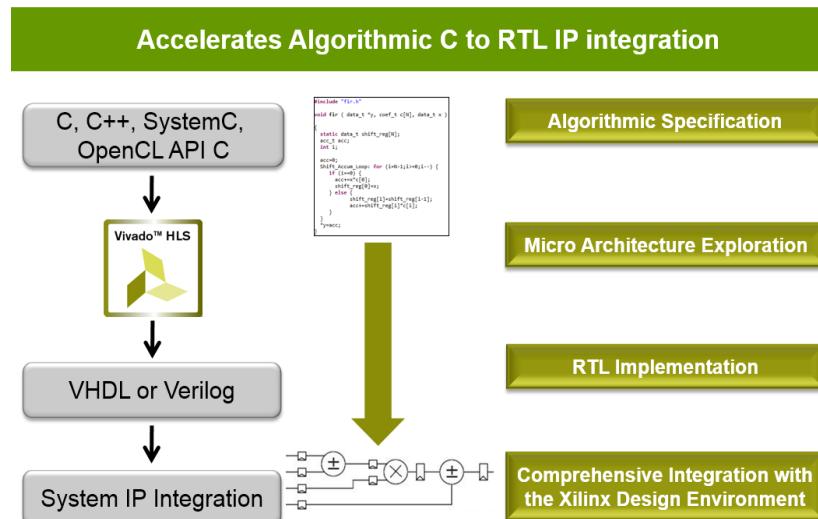
- **Report files:** This output is the result of synthesis, C/RTL co-simulation, and IP packaging.

Slide 2-7:

High-Level Synthesis: HLS



- High-level synthesis
 - Creates an RTL implementation from source code
 - C, C++, SystemC, OpenCL API C kernel
 - Coding style impacts hardware realization
 - Limitations on certain constructs and access to libraries
 - Extracts control and dataflow from the source code
 - Implements the design based on defaults and user-applied directives



- Many implementations are possible from the same source description
 - Smaller designs, faster designs, optimal designs
- Enables design exploration

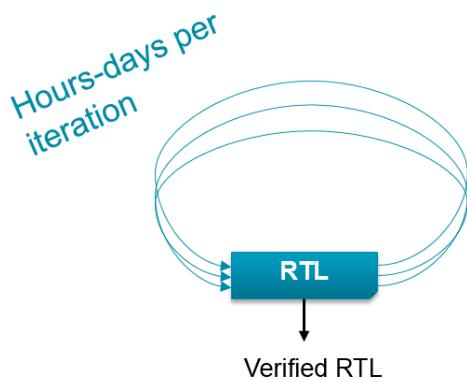
Slide 2-8:

Verification Productivity

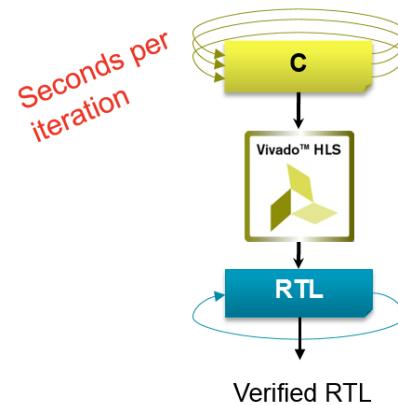
- A C-based design provides much faster verification



HDL-Based Design



C-Based Design



Optical flow video design

Input	RTL Simulation Time	C Simulation Time	Acceleration
10 frames of video data	~2 days	10 seconds	~12,000X



Another technology that will make FPGA design more efficient is C-based high-level synthesis. Using C as the design language can have a huge impact on simulation run times (thousands of times faster) and thereby make the design process much more iterative.

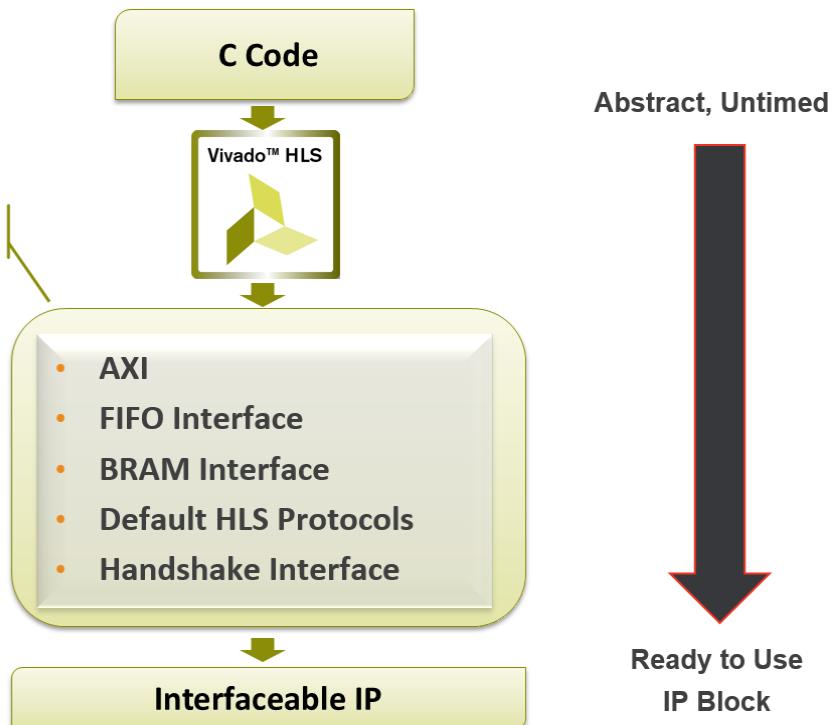
In the video example above, the simulation time was improved 12000x yet no more time was required to verify the final RTL. The Vivado HLS tool dramatically reduces verification effort and time and has been achieving equivalent or better QoR, especially on DSP algorithms.

Slide 2-9:

Vivado HLS Tool – Interfaces

- Vivado HLS tool can use of variety of I/O types

• Directives / Pragmas



From an array or a pointer you can create an AXI4 master or you can make an AXI4-Lite. A master interface can be created in HLS from a C pointer or an array (by a Tcl command or a directive).

Slide 2-10:

Basics of the Vivado HLS Tool



- Introduction to High-Level Synthesis
- **Basics of the Vivado HLS Tool**
- Design Exploration
- Language Support
- Validation Flow
- HLS UltraFast Design Methodology
- Summary

Slide 2-11:

Basics of High-Level Synthesis



- High-level synthesis includes the following phases
 - Scheduling
 - Determines which operations occur during each clock cycle
 - Binding
 - Determines which hardware resource implements each scheduled operation
 - Control logic extraction
 - Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design



Scheduling

Determines which operations occur during each clock cycle based on:

- The length of the clock cycle or clock frequency
- The time it takes for the operation to complete as defined by the target device
- User-specified optimization directives

If the clock period is longer or a faster FPGA is targeted, more operations are completed within a single clock cycle, and all operations might complete in one clock cycle. Conversely, if the clock period is shorter or a slower FPGA is targeted, high-level synthesis automatically schedules the operations over more clock cycles, and some operations might need to be implemented as multicycle resources.

Binding

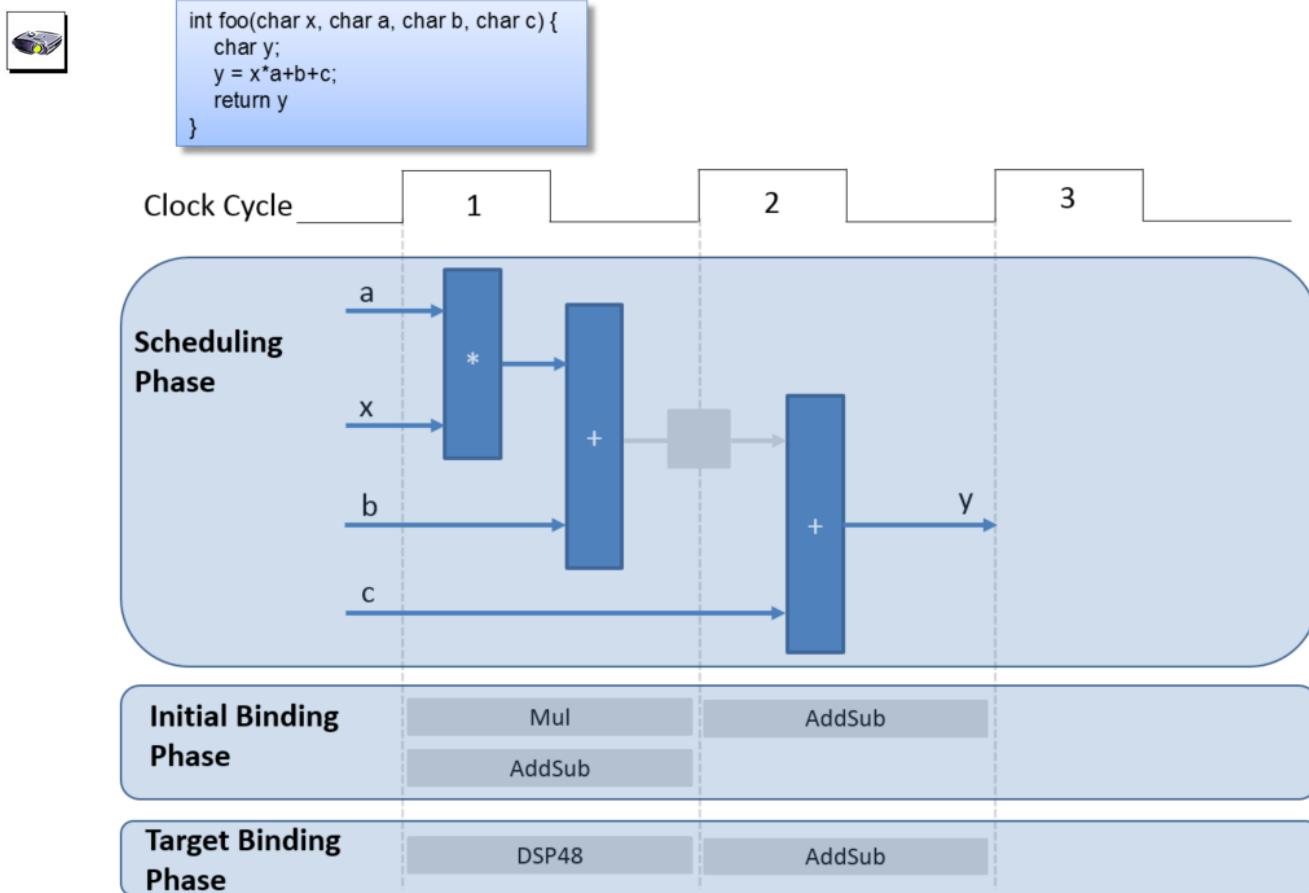
Determines which hardware resource implements each scheduled operation. To implement the optimal solution, high-level synthesis uses information about the target device.

Control Logic Extraction

Extracts the control logic to create a finite state machine that sequences the operations in the RTL design.

Slide 2-12:

Scheduling and Binding Example





In the **scheduling phase** of this example, high-level synthesis schedules the following operations to occur during each clock cycle:

- First clock cycle: Multiplication and the first addition
- Second clock cycle: Second addition and output generation

Note: In the figure above, the square between the first and second clock cycles indicates when an internal register stores a variable. In this example, high-level synthesis only requires that the output of the addition is registered across a clock cycle. The first cycle reads the x, a, and b data ports. The second cycle reads data port c and generates output y.

In the final hardware implementation, high-level synthesis implements the arguments to the top-level function as input and output (I/O) ports. In this example, the arguments are simple data ports. Because each input variables is a char type, the input data ports are all 8-bits wide. The function return is a 32-bit int data type, and the output data port is 32-bits wide.

Important: The advantage of implementing the C code in the hardware is that all operations finish in a shorter number of clock cycles. In this example, the operations complete in only two clock cycles. In a central processing unit (CPU), even this simple code example takes more clock cycles to complete.

In the **initial binding phase** of this example, high-level synthesis implements the multiplier operation using a combinational multiplier (Mul) and implements both add operations using a combinational adder/subtractor (AddSub).

In the **target binding phase**, high-level synthesis implements both the multiplier and one of the addition operations using a DSP48 resource. The DSP48 resource is a computational block available in the FPGA architecture that provides the ideal balance of high performance and efficient implementation.

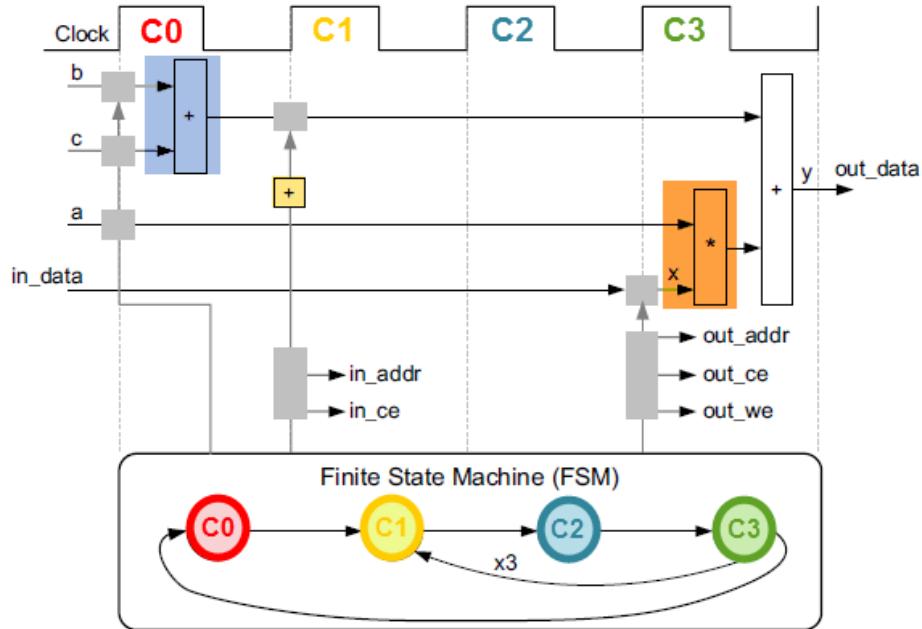
Slide 2-13:

Control Logic Extraction and I/O Port Implementation Example



```
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;
    }
}
```

C0 - adder $b + c$
C1 - generates the address for *in* and *adder* to increment to count how many times design iterates {C1, C2 and C3}
C2 - Block RAM returns the data for *in* and stores it as variable *x*
C3 - Reads the data from port *a* with other values to perform the calculation and generates the first *y* output



This code example performs the same operations as the previous example. However, it performs the operations inside a *for* loop, and two of the function arguments are arrays. The resulting design executes the logic inside the *for* loop three times when the code is scheduled.

High-level synthesis automatically extracts the control logic from the C code and creates an FSM in the RTL design to sequence these operations. High-level synthesis implements the top-level function arguments as ports in the final RTL design.

The scalar variable of type *char* maps into a standard 8-bit data bus port. Array arguments, such as *in* and *out*, contain an entire collection of data.

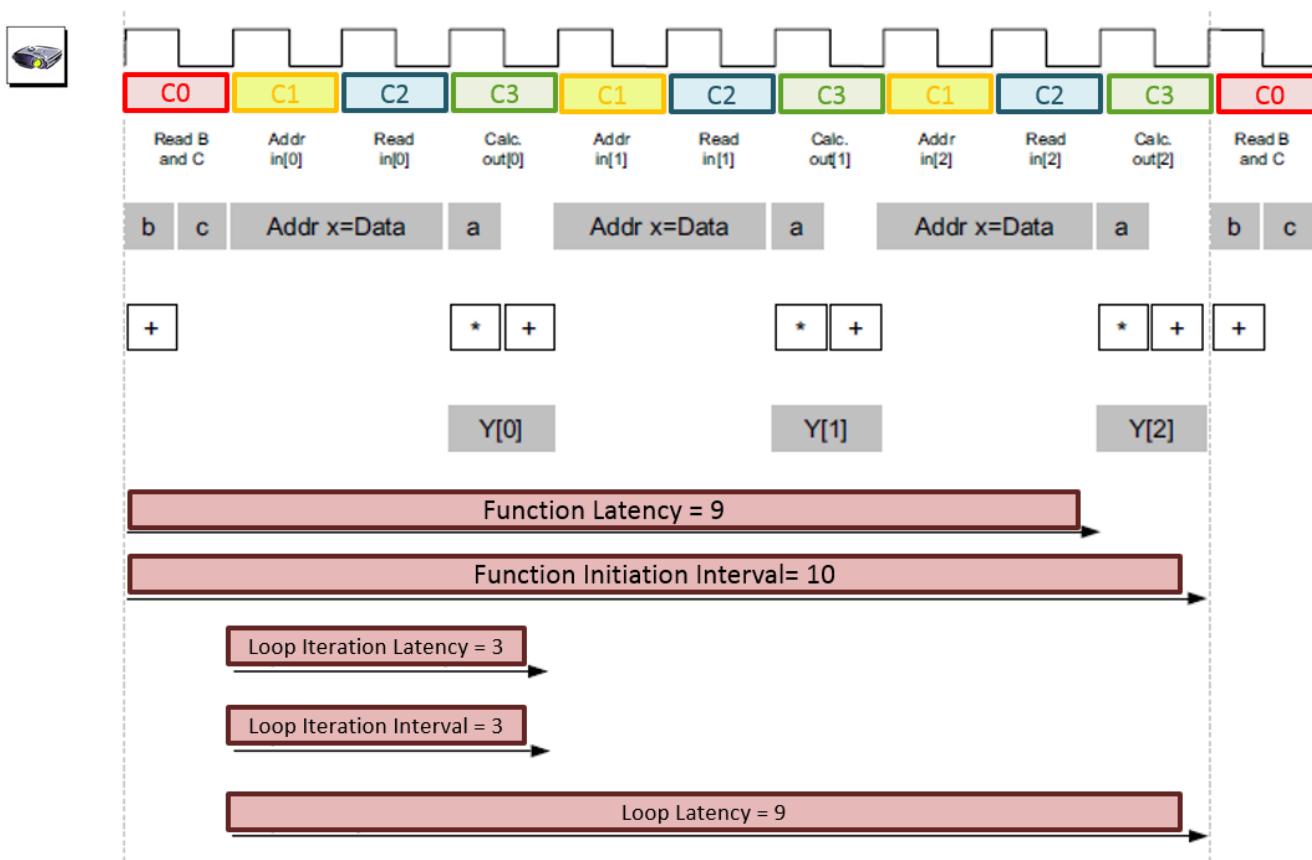
In high-level synthesis, arrays are synthesized into block RAM by default, but other options are possible, such as FIFOs, distributed RAM, and individual registers.

When using arrays as arguments in the top-level function, high-level synthesis assumes that the block RAM is outside the top-level function and automatically creates ports to access a block RAM outside the design, such as data ports, address ports, and any required chip-enable or write-enable signals.

The FSM controls when the registers store data and controls the state of any I/O control signals. The FSM starts in the state C0. On the next clock, it enters state C1, then state C2, and then state C3. It returns to state C1 (and C2, C3) a total of three times before returning to state C0.

Slide 2-14:

Performance Metric Example



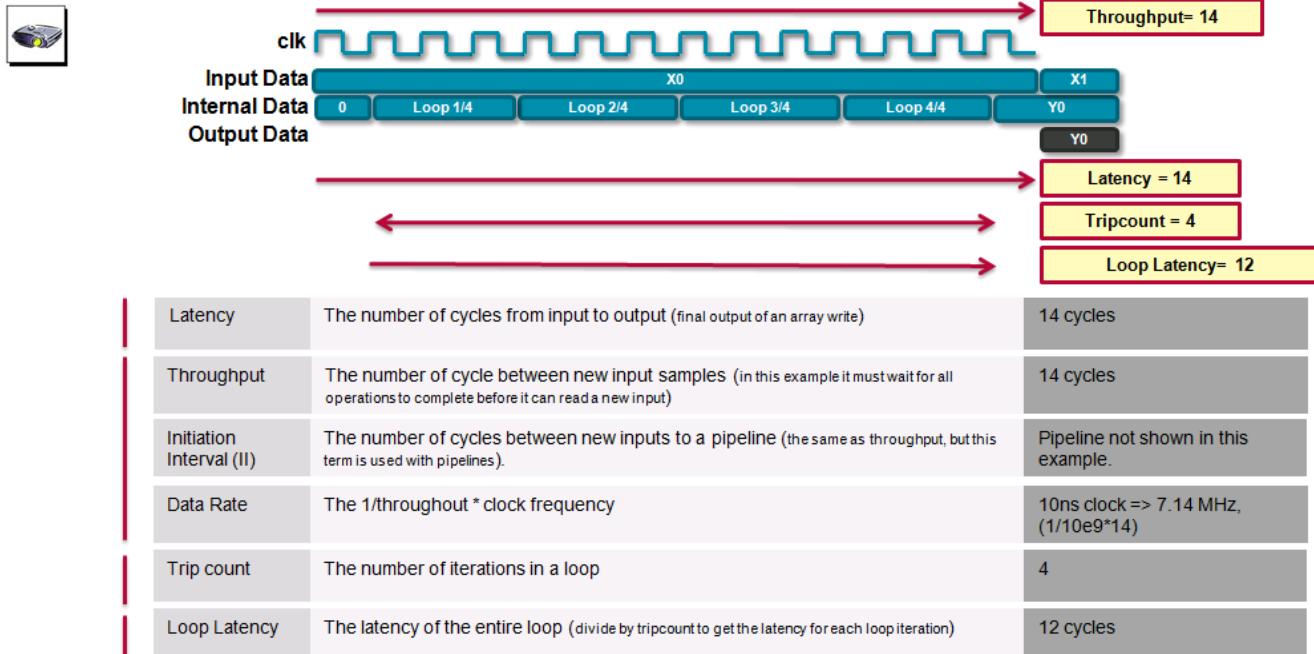
The following are the performance metrics for this example:

- **Latency:** It takes the function nine clock cycles to output all values.
 - When the output is an array, the latency is measured to the last array value output.

- **Initiation Interval (II):** The II is 10, which means it takes 10 clock cycles before the function can initiate a new set of input reads and start to process the next set of input data.
 - The time to perform one complete execution of a function is referred to as one transaction. In this example, it takes 11 clock cycles before the function can accept data for the next transaction.
- **Loop iteration latency:** The latency of each loop iteration is three clock cycles.
- **Loop II:** The interval is three.
- **Loop latency:** The latency is nine clock cycles.

Slide 2-15:

Terminology for Measuring in Clock Cycles



You may have your own terminology – The Vivado HLS tool uses the above



If an array is used on the output, the write of the final data element in the array is used to report the latency.

This example shows an internal loop with four iterations.

Slide 2-16:

Key Attributes of C Code for HLS



- Functions: All code is made up of functions that represent the design hierarchy; the same in hardware
- Top-level I/O: The arguments of the top-level function determine the hardware RTL interface ports
- Types: All variables are of a defined type. The type can influence the area and performance

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



C/C++ code is based on functions—even the *main()* is a function. All functions contain an argument list, which may be empty. The Vivado HLS tool converts each argument to a physical connection. The type of physical connection is defined by the *type* of the argument.

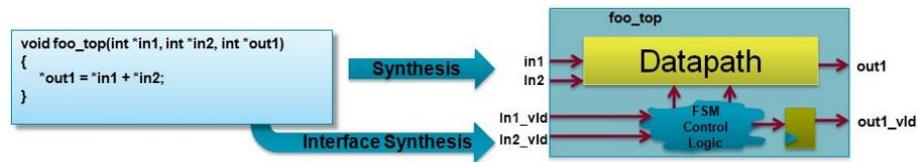
In example above, *x* is defined as a *data_t* type, the array *c* is defined as a *coef_t* type, etc. These predetermined types are supported by the Vivado HLS tool, which knows the proper way to construct hardware from these types.

Slide 2-17:

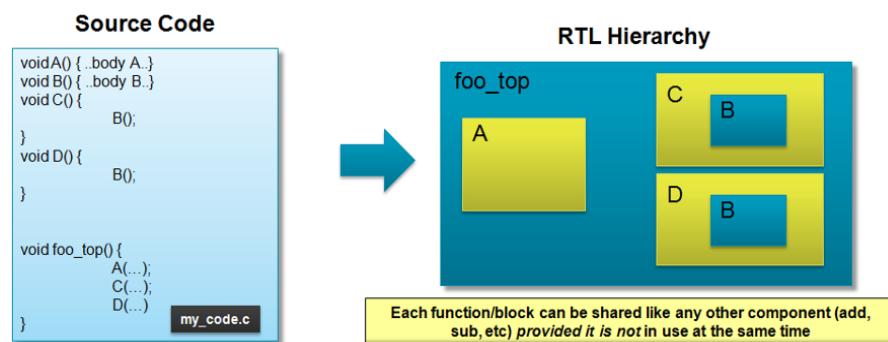
C to RTL Conversion



- HLS synthesizes the C code in different ways



- Top-level function arguments synthesize into RTL I/O ports
- C functions synthesize into blocks in the RTL hierarchy



- Loops in the C functions are kept *rolled* by default
- Arrays in the C code synthesize into block RAM in the final design



Top-level function arguments:

- Become ports on the RTL designs
- Can optimally be implemented with a hardware protocol: interface synthesis
- Additional control ports are added to the design

I/O protocols:

- Allow RTL blocks to automatically synchronize data exchange
- Many type of protocols are available

Array can be targeted to any memory resource in the library:

- Ports (address, CE active high, etc.) and sequential operation (clocks from address to data out) are defined by the library model
- Arrays can be partitioned into individual elements
 - Implemented as smaller RAMs or registers

Slide 2-18:

Design Exploration

- Introduction to High-Level Synthesis
- Basics of the Vivado HLS Tool
- Design Exploration**
- Language Support
- Validation Flow
- HLS UltraFast Design Methodology
- Summary

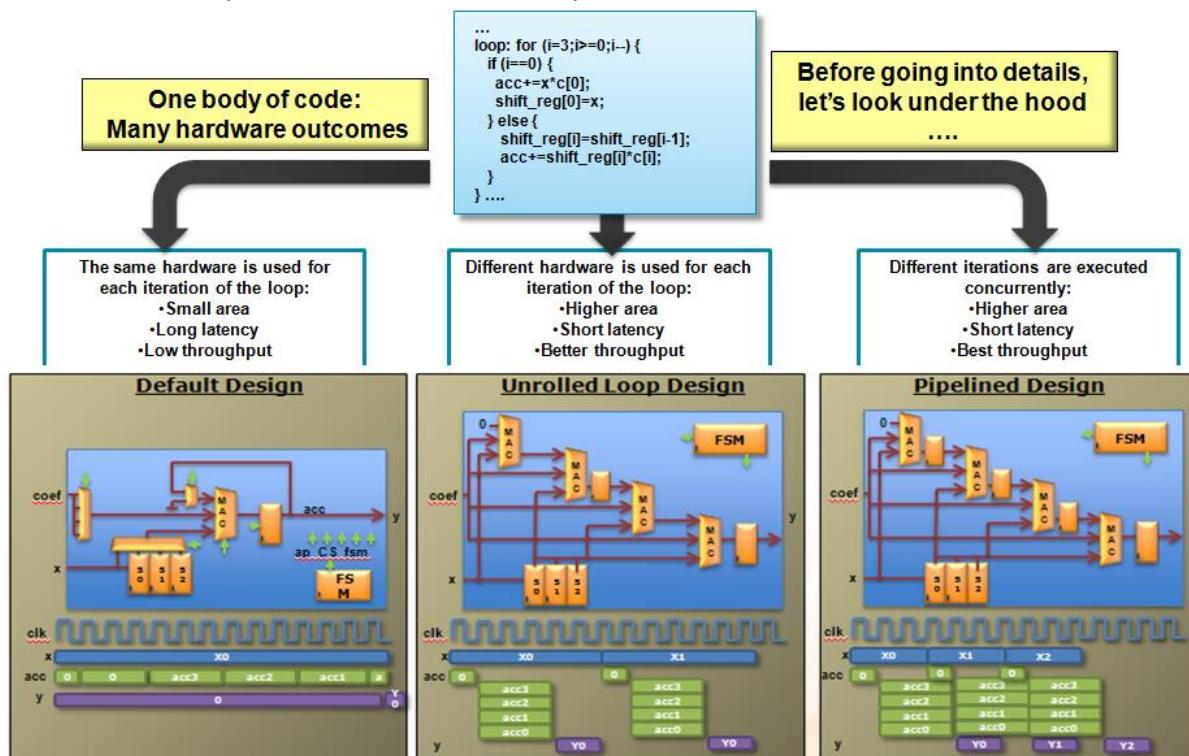


Slide 2-19:

Design Exploration with Pragmas



- One C code can have multiple hardware implementations
- Small changes to pragmas change the resulting hardware
 - Use to explore tradeoffs between performance and area



Slide 2-20:

Initiation Interval (II)



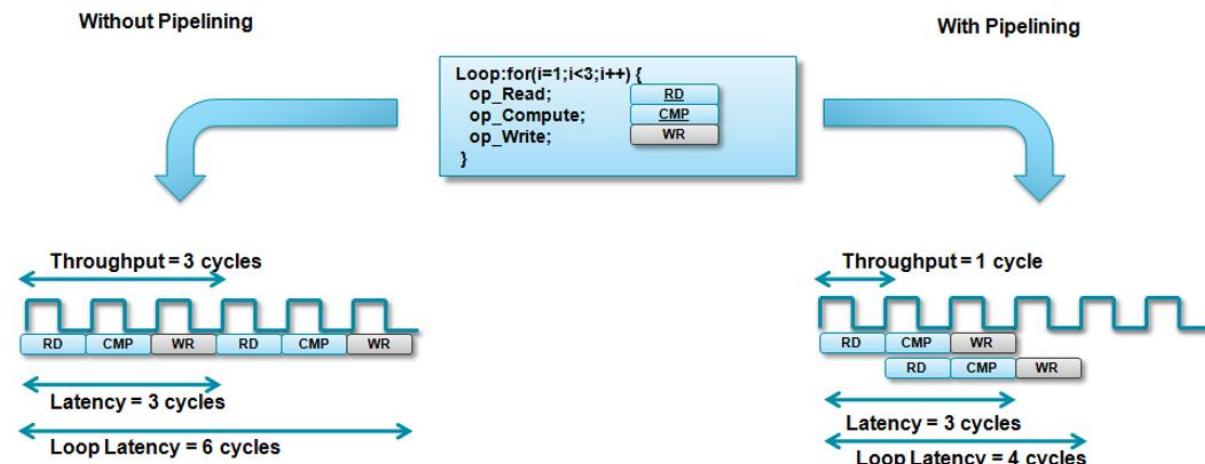
- The initiation interval (II) is the number of clock cycles between new input samples



- The ultimate goal of optimization with HLS is to achieve initiation interval (II)

Slide 2-21:

Optimizing the Loop: Pipelining



- Runs sequentially
- Throughput = three clock cycles
- Latency
 - Three cycles per iteration
 - Six cycles for entire loop
- Runs in pipelined manner
- Throughput = one clock cycle
- Latency
 - Three cycles per iteration
 - Four cycles for entire loop

Slide 2-22:

Arrays and Memory Bottlenecks



- Arrays are the basic construct to express memory to HLS
- Default number of memory ports is defined by
 - Number of usages in the algorithm
 - Target throughput
- HLS default memory model assumes dual-port block RAM
- Arrays can be reshaped and partitioned to remove bottlenecks
 - Changes to array layout *does not* require changes to original code

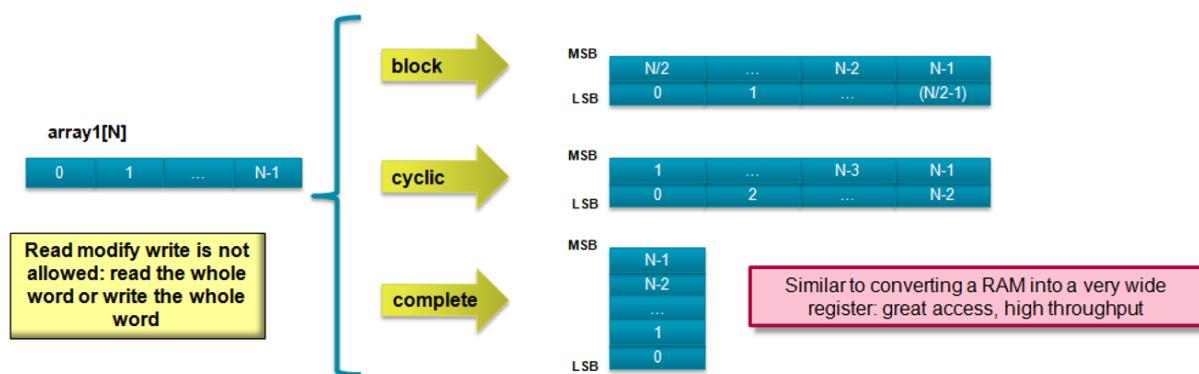


Slide 2-23:

Data Access



- Increase the data bandwidth by reshaping the input array
- No change to the original C code by use of synthesis directives
- Be aware of I/O size



Slide 2-24:

Strategies for a Big Design

- For designs with multiple functions



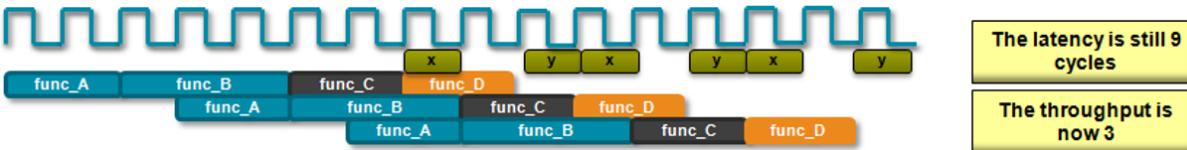
```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(a,b,t1);
    func_B(a,t1,t2);
    func_C(c,t2,&x)
    func_D(d,x,&y)
}
```



- Default scheduling



- Functions can be pipelined using dataflow directives



Slide 2-25:

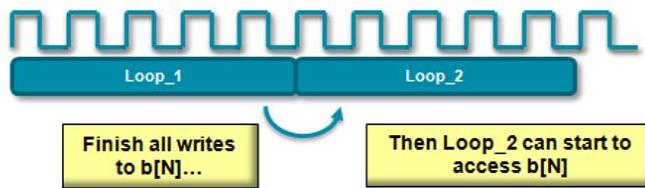
Dataflow

- Dataflow is the parallel execution of multiple loops within a function

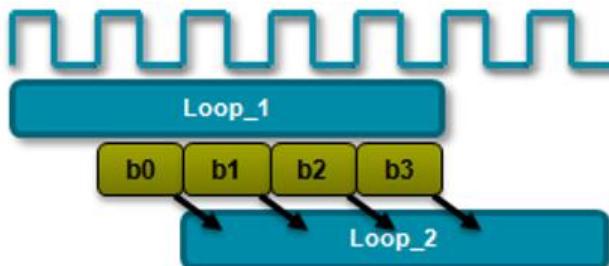


- Loops that run in parallel communicate through arrays

```
int a[N], b[N], c[N];
Loop_1: for (i=0;i<=N-1;i++) {
    b[i] = a[i] + in1;
}
Loop_2: for (i=0;i<=N-1;i++) {
    c[i] = b[i] * in2;
}
```

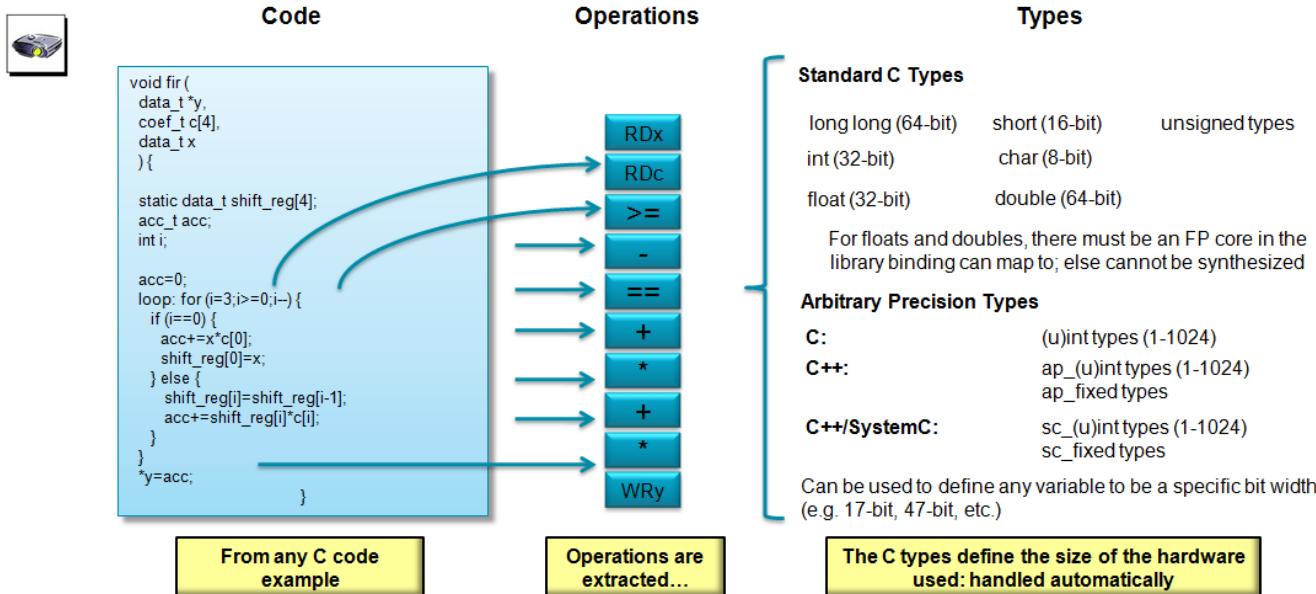


- Arrays are changed to FIFOs to allow concurrent execution of Loop_1 and Loop_2



Slide 2-26:

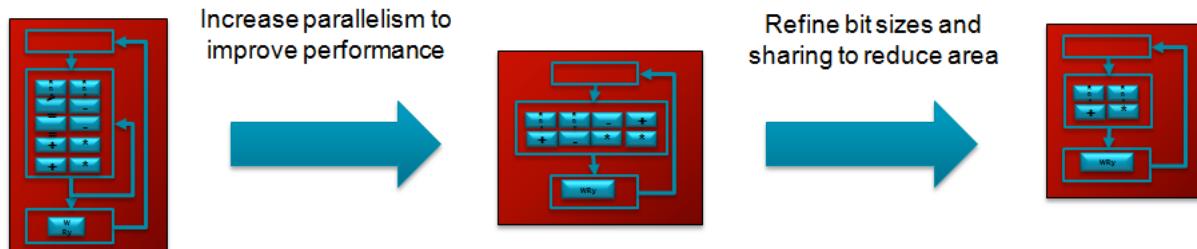
Types = Operator Bit-sizes



Slide 2-27:

High-Level Synthesis (HLS)

- In summary, HLS design involves
 - Synthesizing the initial design
 - Analyzing to see what limits the performance
 - User directives to change the default behaviors
 - Removing bottlenecks
 - Analyzing to see what limits the area
 - The types used define the size of operators
 - This can have an impact on what operations can fit in a clock cycle
- Use directives to shape the initial design to meet performance



Slide 2-28:

Language Support

- Introduction to High-Level Synthesis
- Basics of the Vivado HLS Tool
- Design Exploration
- **Language Support**
- Validation Flow
- HLS UltraFast Design Methodology
- Summary



Slide 2-29:

Comprehensive C Support



- A complete C validation and verification environment
 - Vivado HLS tool supports complete bit-accurate validation of the C model
 - Vivado HLS tool provides a productive C-RTL co-simulation verification solution
- Vivado HLS tool supports C, C++, SystemC, and OpenCL API C Kernel
 - Functions can be written in any version of C
 - Wide support for coding constructs in all three variants of C
 - Easier to discuss what is not supported than what is



Starting with Vivado Design Suite 2015.1, the Vivado HLS tool supports OpenCL API C language constructs and built-in functions from the OpenCL API C 1.0 embedded profile.

The Vivado HLS tool supports the following standards for C compilation/simulation:

- ANSI-C (GCC 4.6)
- C++ (G++ 4.6)
- OpenCL API (1.0 embedded profile)
- SystemC (IEEE 1666-2006, version 2.2)

Slide 2-30:

Library Support (1)



- Arbitrary Precision Data Types library
 - Support for both integer (*ap_cint.h*, *ap_int.h*) and fixed-point (*ap_fixed.h*) arbitrary precision data types
- HLS MATH library (*hls_math.h*)
 - Extensive support for the synthesis of the standard C (*math.h*) and C++ (*cmath.h*) libraries
 - Support includes floating-point and fixed-point functions
- HLS Stream library (*hls_stream.h*)
 - Provides a C++ template class `hls::stream<>` for modeling streaming data structures
- HLS Video library (*hls_video.h*)
 - OpenCV video function support
 - Libraries target real-time, full HD video processing



You can use each of the C libraries in your design by including the library header file. These header files are located in the include directory in the Vivado HLS tool installation area.

Note: The header files for the Vivado HLS tool C libraries do not have to be in the include path if the design is used in the Vivado HLS tool. The paths to the library header files are automatically added.

Slide 2-31:

Library Support (2)



- HLS IP library
 - Instantiates and parameterizes the FIR compiler (*hls_fir.h*) and FFT LogiCORE™ IP (*hls_fft.h*) as function calls from your C++ code
 - Implements a shift register using a Xilinx SRL primitive (*ap_shift_reg.h*)
 - Implements a Xilinx direct digital synthesizer (DDS) from your a C++ code (*hls_dds.h*)
- HLS Linear Algebra library (*hls_linear_algebra.h*)
 - Provides a number of commonly used linear algebra functions
- HLS DSP library



HLS IP libraries:

- *hls_fft.h*: Allows the Xilinx LogiCORE IP FFT to be simulated in C and implemented using the Xilinx LogiCORE block.
- *hls_fir.h*: Allows the Xilinx LogiCORE IP FIR 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 that is implemented directly using a Xilinx SRL primitive.

Slide 2-32:

C, C++, and SystemC Support



- Vast majority of C, C++, and SystemC is supported
 - Provided it is statically defined at **compile time**
 - If a function or variable is not fully realized until it would normally be executed
 - It cannot be synthesized
 - That is, anything that relies on dynamic memory allocation
- Any of the three variants of C can be used
 - If C is used, the Vivado HLS tool expects the file extensions to be .c
 - For C++ and SystemC, it expects file extensions .cpp

Slide 2-33:

Unsupported Constructs: Overview



- System calls and function pointers
 - Dynamic memory allocation
 - *malloc()*, *alloc()*, and *free()*
 - Standard I/O 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



Slide 2-34:

Validation Flow



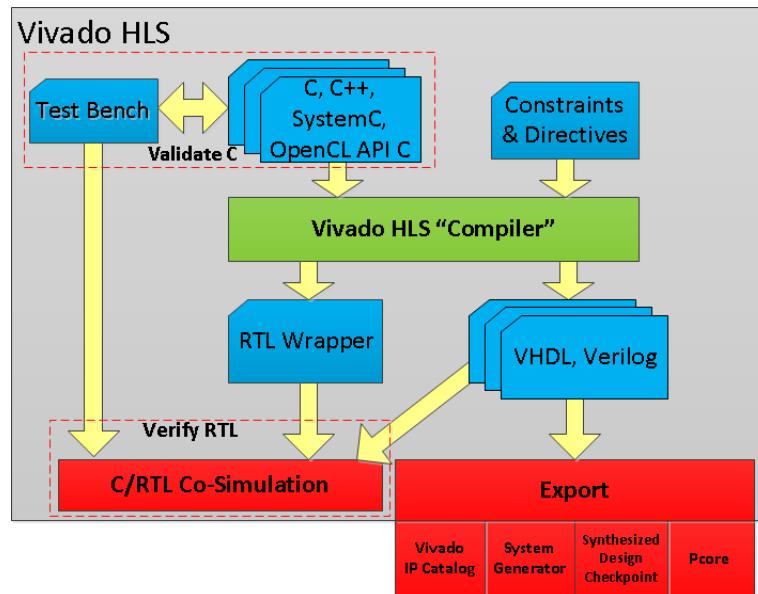
- Introduction to High-Level Synthesis
- Basics of the Vivado HLS Tool
- Design Exploration
- Language Support
- **Validation Flow**
- HLS UltraFast Design Methodology
- Summary

Slide 2-35:

C Validation and RTL Verification



- Two steps to verifying the design
 - Pre-synthesis: C **validation**
 - Validate the algorithm is correct
 - Post-synthesis: RTL **verification**
 - Verify the RTL is correct



- C validation
 - A HUGE advantage to using HLS
 - Fast, free verification
 - Validating the algorithm is correct **before** synthesis
 - Follow the given testbench tips
- RTL verification
 - Vivado HLS tool can co-simulate the RTL with the original testbench

Slide 2-36:

C Function Testbench



- Testbench is the level above the function
 - The *main()* function is **above** the function to be synthesized
- Good practices
 - Testbench should compare the results with golden data
 - Automatically confirms any changes to the C are validated
 - Automatically verifies the RTL is correct
 - Testbench should return a 0 if the self-checking is correct
 - Anything but a 0 (zero) will cause RTL verification to issue a FAIL message
 - Function *main()* should expect an integer return (non void)

```
int main () {  
    int ret=0;  
    ...  
    ret = system("diff --brief -w output.dat output.golden.dat");  
  
    if (ret != 0) {  
        printf("Test failed !!!\n");  
        ret=1;  
    } else {  
        printf("Test passed !\n");  
    }  
    ...  
    return ret;  
}
```

Slide 2-37:

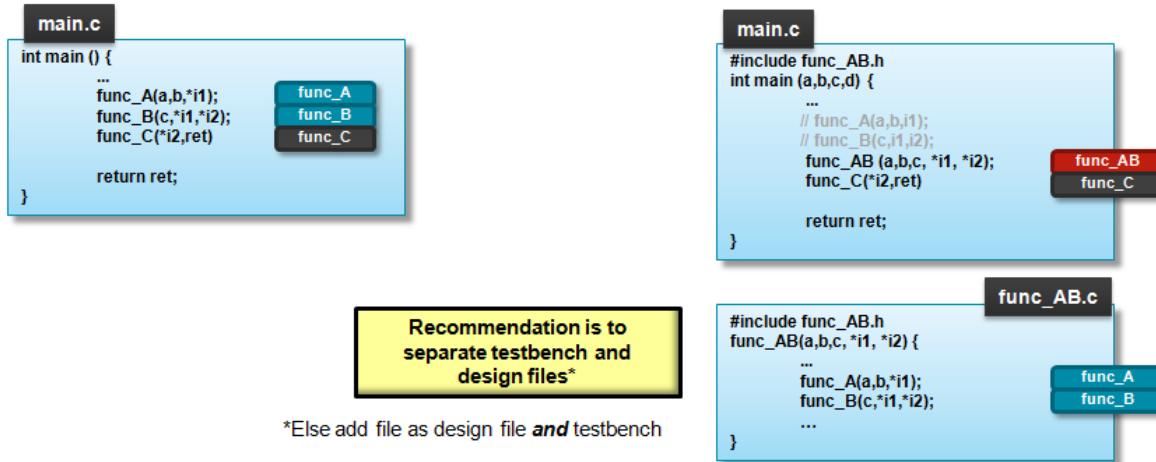
Determine or Create the Top-Level Function



- Determine the top-level function for synthesis
- If there are multiple functions, they must be merged
 - There can only be one top-level function for synthesis

Given a case where functions func_A and func_B are to be implemented in FPGA

Re-partition the design to create a new single top-level function inside main()



Slide 2-38:

Validation Flow

- Introduction to High-Level Synthesis
- Basics of the Vivado HLS Tool
- Design Exploration
- Language Support
- Validation Flow
- **HLS UltraFast Design Methodology**
- Summary



Slide 2-39:

HLS UltraFast Design Methodology (1)



- C testbench
 - Verify the design before synthesizing
 - Should be a self-checking testbench
- Language constructs
 - Verify the design for VHLS unsupported constructs and change them as needed
- Understand the concurrent hardware
 - To ensure that high-performance C operation be performed in parallel and executed in a pipelined manner
- Synthesis strategies
 - Run the baseline synthesis to understand what is happening
 - Create new runs and apply directives to implement solutions that you are expecting

Slide 2-40:

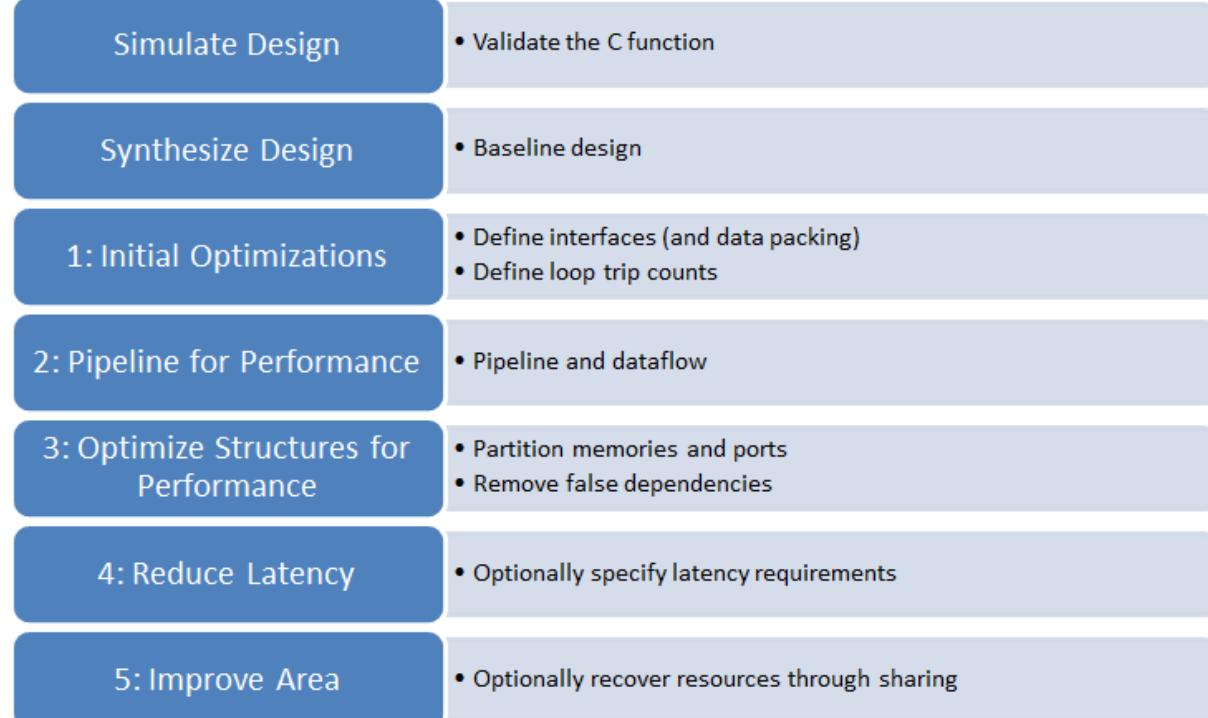
HLS UltraFast Design Methodology (2)



- Writing hardware efficient C code
 - Understand the differences for coding CPU (and/or GPU) vs. FPGA
- Data types for efficient hardware
 - Make use of arbitrary precision data types to implement the design to save resources and get high performance
- Using hardware-optimized C libraries
 - Make use of the Vivado HLS tool-provided C libraries for commonly used C functions to achieve high performance, which results in efficient implementation
- Design analysis and optimization
 - Verify the reports
 - Use the analysis viewer to understand what is happening
 - Make use of compare reports to identify results of different solutions

Slide 2-41:

HLS Optimization Methodology



Slide 2-42:

Summary

- Introduction to High-Level Synthesis
- Basics of the Vivado HLS Tool
- Design Exploration
- Language Support
- Validation Flow
- HLS UltraFast Design Methodology
- **Summary**



Slide 2-43:

Summary (1)



- In high-level synthesis (HLS)
 - C becomes RTL
 - Operations in the code map to hardware resources
 - Understand how constructs such as functions, loops, and arrays are synthesized
- Use directives to shape the initial design to meet performance
 - Increase parallelism to improve performance
 - Refine bit sizes and sharing to reduce area

Slide 2-44:

Summary (2)



- HLS design involves
 - Synthesizing the initial design
 - Analyzing to see what limits the performance
 - User directives to change the default behaviors
 - Remove bottlenecks
 - Analyzing to see what limits the area
 - Types used define the size of operators
 - This can have an impact on what operations can fit in a clock cycle

Capture Your Notes Here



Using the Vivado HLS Tool

Slide 3-1:

2016.1

This module describes the Vivado® HLS tool flow. 60 minutes.

Slide 3-2:

Objectives



After completing this module, you will be able to:

- Identify the steps involved in creating a project using the Vivado HLS tool
- Describe the C development environment in the Vivado HLS tool
- Discuss the synthesis and implementation steps in the Vivado HLS tool
- Identify the functionality of the Design viewer analysis
- Describe the Vivado HLS tool RTL verification flow
- Discuss the recommended implementation flow

Slide 3-3:

Invoking the Vivado HLS Tool

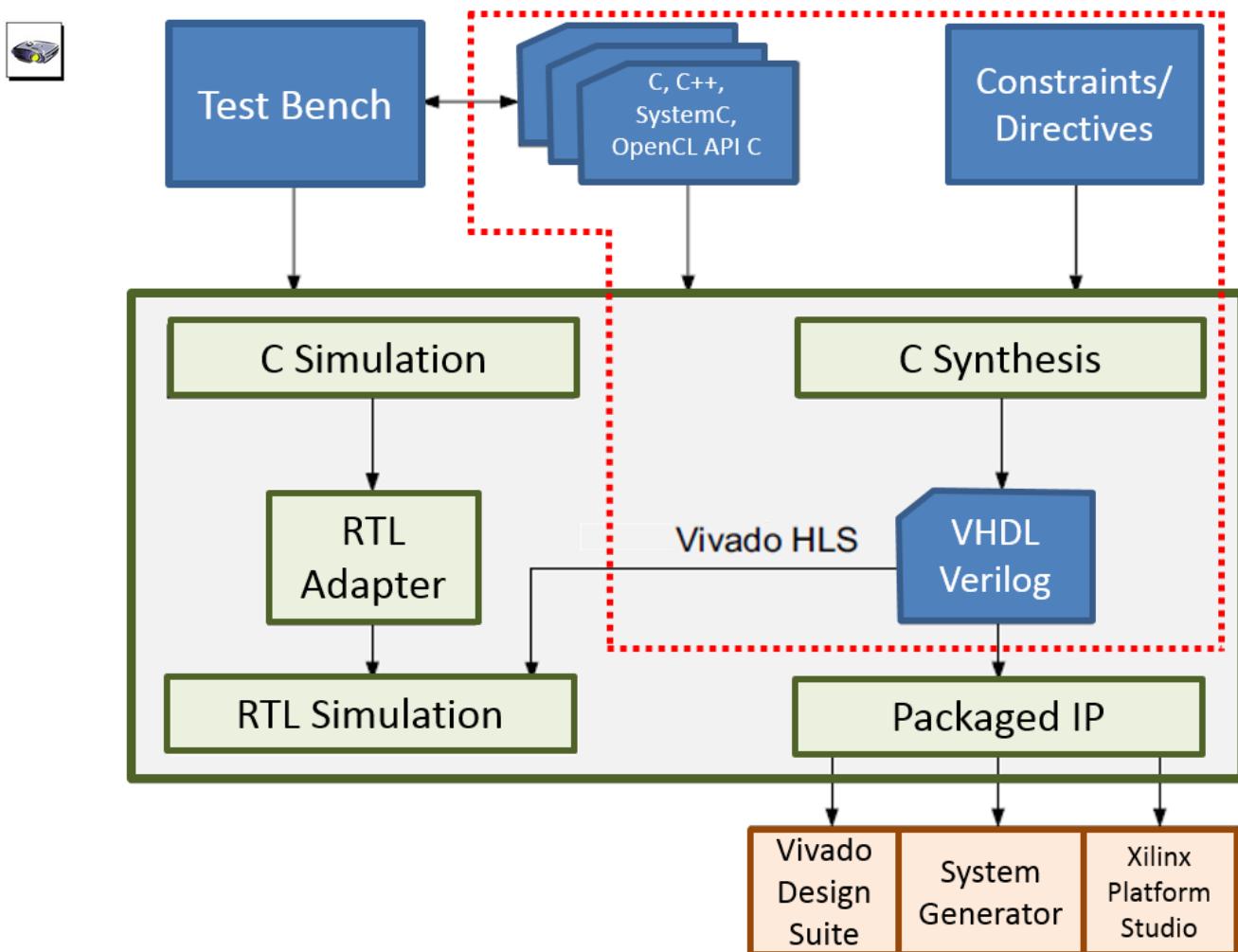


▪ Invoking the Vivado HLS Tool

- Projection Creation in the Vivado HLS Tool
- C Validation to IP Creation
- Adding Directives
- Vivado HLS Tool Directory Structure
- Summary

Slide 3-4:

Vivado HLS Tool: The Synthesis Process





The figure above shows the overall structure of HLS and its high-level components and flow. The synthesis portion of the design is highlighted by the red dotted line: Inputs to the Vivado HLS tool *compiler* are the sources for synthesis and any constraints and/or directive files.

The *compiler* uses the constraints and directive files to determine how the HDL output should be generated. The outputs are HDL files that can then be simulated and exported in several different flavors for use by a number of different tools.

Slide 3-5:

Vivado HLx Tool OS Support



- Vivado HLS tool is free of charge and part of all Vivado Design Suite Editions
- Vivado Design Suite HLx Editions
 - Vivado HL System Edition
 - Vivado HL Design Edition
 - Vivado HL WebPACK™ Edition
- Vivado HLx tool is supported on both Linux and Windows



Vivado Design Suite 2015.4 is the last release of the Vivado HLS tool that allows targeting of devices that are only supported by the ISE® Design Suite.

IMPORTANT: This means that you will need to archive Vivado Design Suite 2015.4 in order to target devices such as Spartan®-6 or Virtex®-6 FPGAs as future releases of the Vivado HLS tool will only support devices supported by the main Vivado IDE.

Slide 3-6:

Device Support

	Vivado WebPACK Tool	Vivado Design Suite (All Other Editions)
Zynq® Device	Zynq®-7000 AP SoC Device XC7Z010, XC7Z015, XC7Z020, XC7Z030	Zynq®-7000 AP SoC Device - All
Virtex® FPGA	Virtex-7 FPGA - None Virtex UltraScale FPGA - None	Virtex-7 FPGA - All Virtex UltraScale FPGA - All
Kintex® FPGA	Kintex-7 FPGA - XC7K70T, XC7K160T Kintex UltraScale FPGA - XCKU025, XCKU035	Kintex-7 FPGA - All Kintex UltraScale FPGA - All
Artix® FPGA	Artix-7 FPGA - XC7A15T, XC7A35T, XC7A50T, XC7A75T, XC7A100T, XC7A200T	Artix-7 FPGA - All



Vivado Design Suite 2015.4 is the last release of the Vivado HLS tool that allows targeting of devices that are only supported by the ISE® Design Suite.

IMPORTANT: This means that you will need to archive Vivado Design Suite 2015.4 in order to target devices such as Spartan®-6 or Virtex®-6 FPGAs as future releases of the Vivado HLS tool will only support devices supported by the main Vivado IDE.

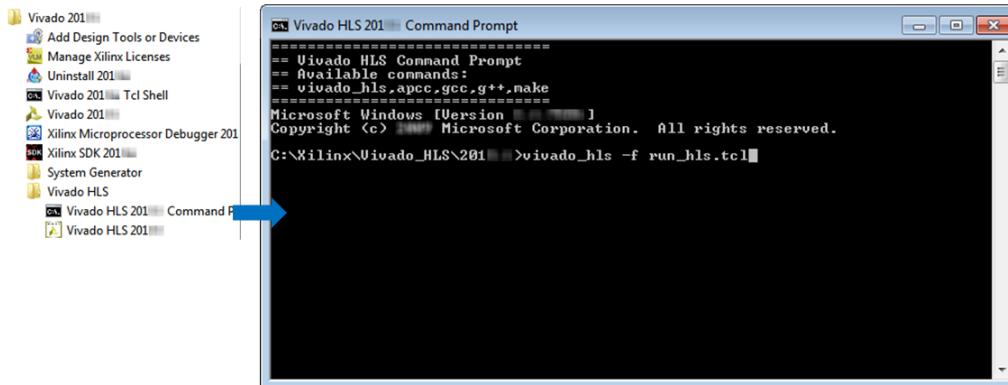
Slide 3-7:

Invoking the Vivado HLS Tool

- Launch the Vivado HLS tool GUI



- Vivado HLS tool can also be run in batch mode



Slide 3-8:

Project Creation in the Vivado HLS Tool



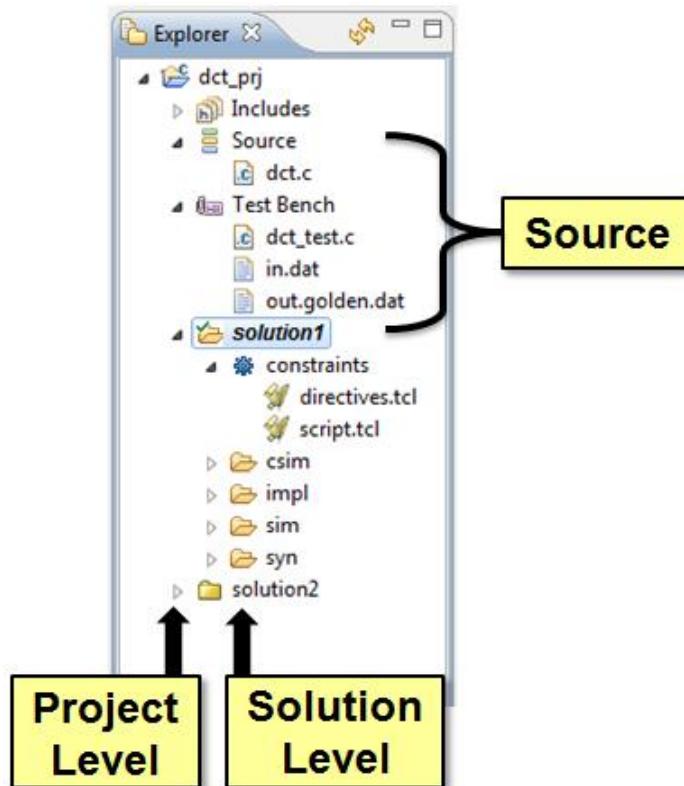
- Invoking the Vivado HLS Tool
- **Projection Creation in the Vivado HLS Tool**
- C Validation to IP Creation
- Adding Directives
- Vivado HLS Tool Directory Structure
- Summary

Slide 3-9:

Vivado HLS Tool Projects and Solutions (1)



- Vivado HLS tool is project based
 - Project specifies the source code that will be synthesized
 - Each project is based on one set of source code
 - Each project has a user-specified name
- Project can contain multiple solutions
 - Solutions are different implementations of the same code
 - Auto-named solution1, solution2, etc.
 - Supports user-specified names
 - Solutions can have different clock frequencies, target technologies, synthesis directives

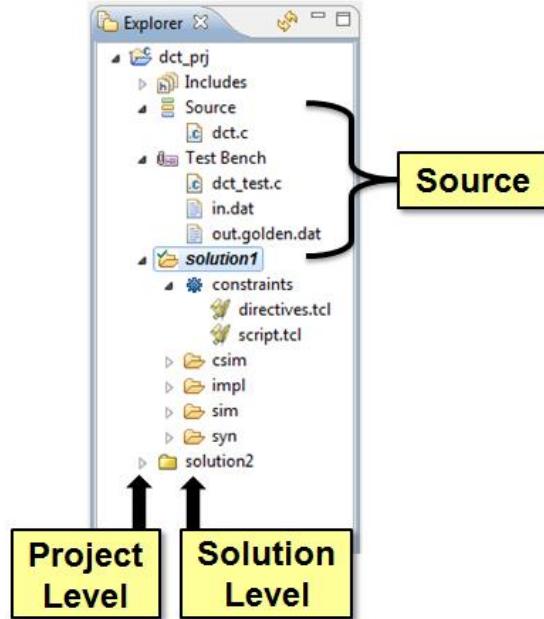


Slide 3-10:

Vivado HLS Tool Projects and Solutions (2)



- Projects and solutions are stored in a hierarchical directory structure
 - Top level is the project directory
 - Disk directory structure is identical to the structure shown in the GUI Explorer tab (except for source code location)

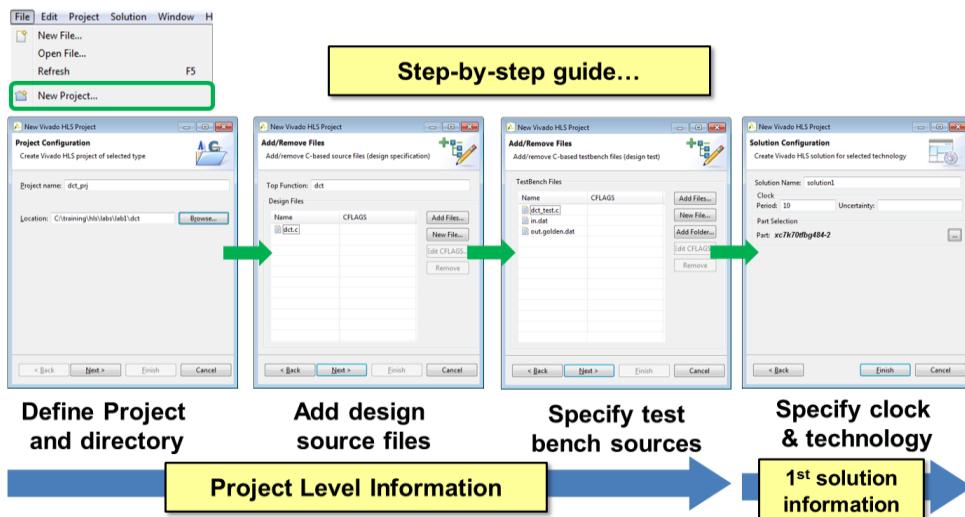


Slide 3-11:

Project Wizard



- Project Wizard guides you through the steps of opening a new project



- Equivalent command line

vivado_hls -p my.prj	add_files [options] <src_files>	add_files tb <src_files>	create_clock -period <number> [OPTIONS] set_clock_uncertainty <uncertainty> <clock_list>
-------------------------	---------------------------------------	-----------------------------	--



The following are some example tasks using the command prompt and Tcl Interface.

Vivado HLS in Interactive Mode: (on Windows and Linux, using the `-i` option opens the Vivado Design Suite in interactive mode)

```
$ vivado_hls -i [-l <log_file>]  
vivado_hls>
```

By default, Vivado HLS creates a `vivado_hls.log` file in the current directory. To specify a different name for the log file, the `-l <log_file>` option can be used.

Executing the Vivado HLS Tool Using a Tcl Batch File: (allows multiple runs to be scripted and automated)

```
vivado_hls -f run_hls.tcl
```

Adding Design Source Files to the Current Project:

Syntax: `add_files [OPTIONS] <src_files>`

where `<src_files>` lists source files with the description of the design.

[OPTIONS] :

- `-tb`: Specifies any files used as part of the design test bench.
- `-cflags <string>`: A string with any desired GCC compilation options.

Creating a Virtual Clock for the Current solution:

Syntax: `create_clock -period <number> [OPTIONS]`

[OPTIONS] :

- `-name <string>`: Specifies the clock name. If no name is given, a default name is used.
- `-period <number>`: Specifies the clock period in ns or MHZ.

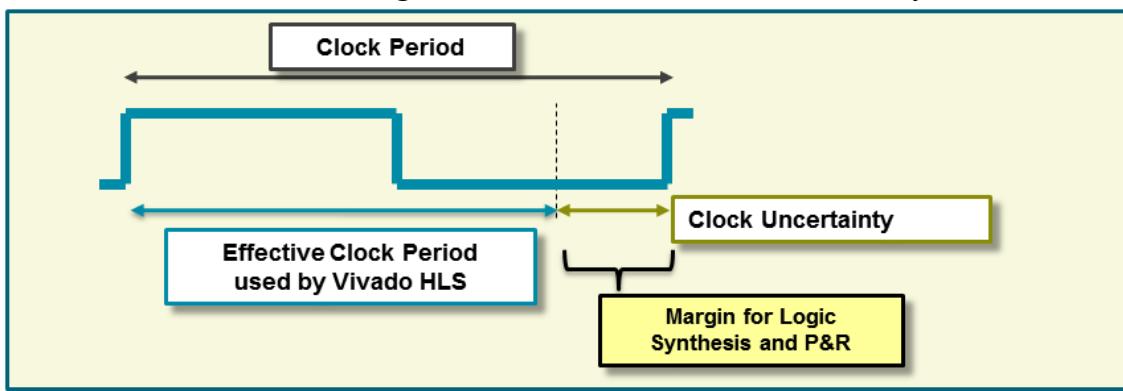
For more information on commands, refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902), Chapter 4: High-Level Synthesis Reference Guide.

Slide 3-12:

Clock Specification



- Clock frequency must be specified
 - Only one clock can be specified for C/C++ functions
 - SystemC can define multiple clocks: one for each sc_cthread
- Clock uncertainty can be specified (leaving it blank / unspecified will default to 12.5% of the clock period)
 - Subtracted from the clock period to give an effective clock period
 - Effective clock period is used for synthesis
 - Should not be used as a design parameter
 - Do not vary for different results: this is your safety margin
 - A user controllable margin to account for downstream RTL synthesis and P&R

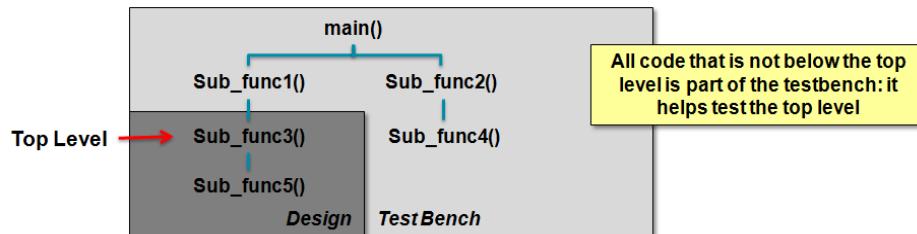


Slide 3-13:

Testbenches (1)



- "Testbench" is any code above the top level
 - Top of any C function is function main()
 - Function main() cannot be the top level for synthesis



- Project files
 - Ideally the testbench and the design are in different files
 - Add the top level as a design file: *add_files my_top.c*
 - Add the testbench as a testbench file: *add_files -tb my_testbench.c*
 - If they are in the same file, add the same file as both a design file and testbench file

Slide 3-14:

Testbenches (2)



- Ideal testbench
 - Should be self-checking
 - RTL verification will re-use the C testbench
 - If the testbench is self-checking
 - Allows RTL verification to be run without a requirement to check the results again
 - RTL verification "passes" if the testbench return value is 0 (zero)
 - Actively return a 0 if the simulation passes
 - If the testbench does not return 0, RTL verification will issue a "fail" message

```
int main () {
    // Test function

    // Compare results
    int ret = system("diff -brief -w test_data/output.dat test_data/golden.dat");

    if (ret != 0) {
        printf("Test failed !!!\n", ret); return 1;
    } else {
        printf("Test passed !!\n", ret); return 0;
    }
}
```

The testbench should return 0 if the results are correct, else return some other value

Slide 3-15:

Synthesis Macro



- Macro `_SYNTHESIS_` is auto defined
 - Defined when Vivado HLS tool elaborates designs
 - Not defined inside Vivado HLS tool when C compilation is performed
 - This can be used to remove unsynthesizable code from the design

```
// test.c
#include <stdio.h>

void test(int d[10]) {
    int acc = 0;
    int i;
#ifndef _SYNTHESIS_
FILE *fp1;
#endif

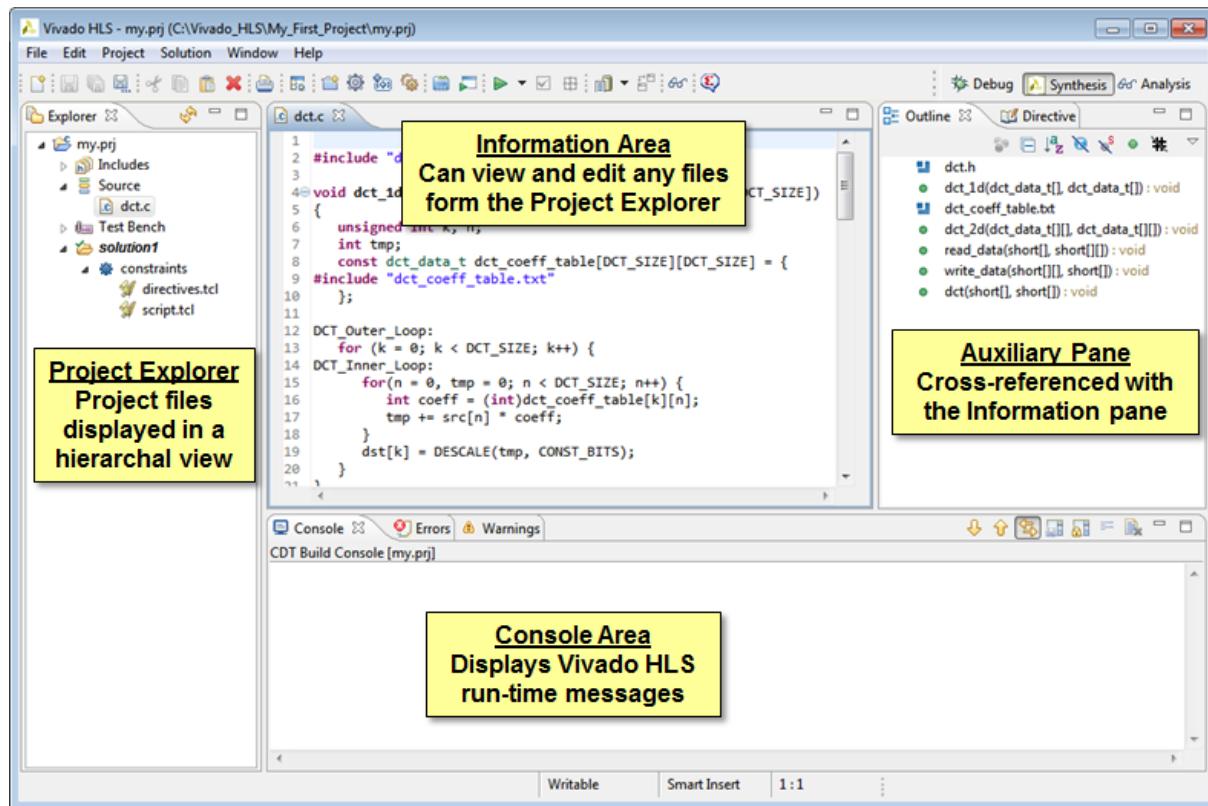
    for (i=0;i<10;i++) {
        acc += d[i];
        d[i] = acc;
#ifndef _SYNTHESIS_
        fprintf(fp1,"%d\n", acc);
#endif
    }
}
```

File access to the OS is ignored during synthesis

Code is only executed if `_SYNTHESIS_` is not defined

Slide 3-16:

Vivado HLS Tool Project

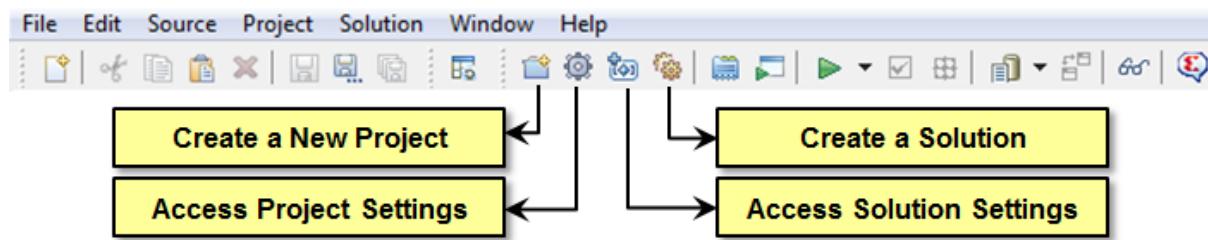


Slide 3-17:

Vivado HLS Toolbar: Project and Solution Control



- Primary commands have toolbar buttons
 - Allows easy access for the most common functions
 - Project and solution controls

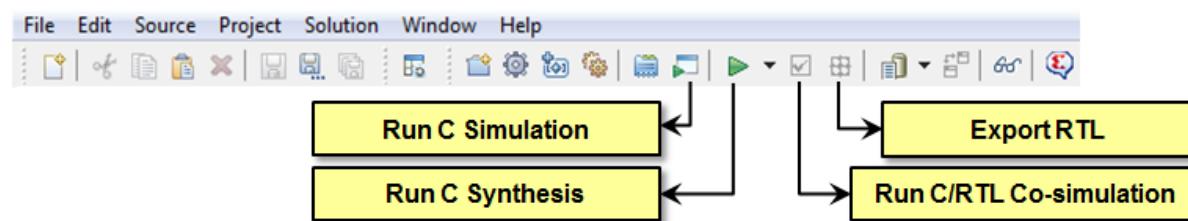


Slide 3-18:

Vivado HLS Toolbar: Execute and Analyze

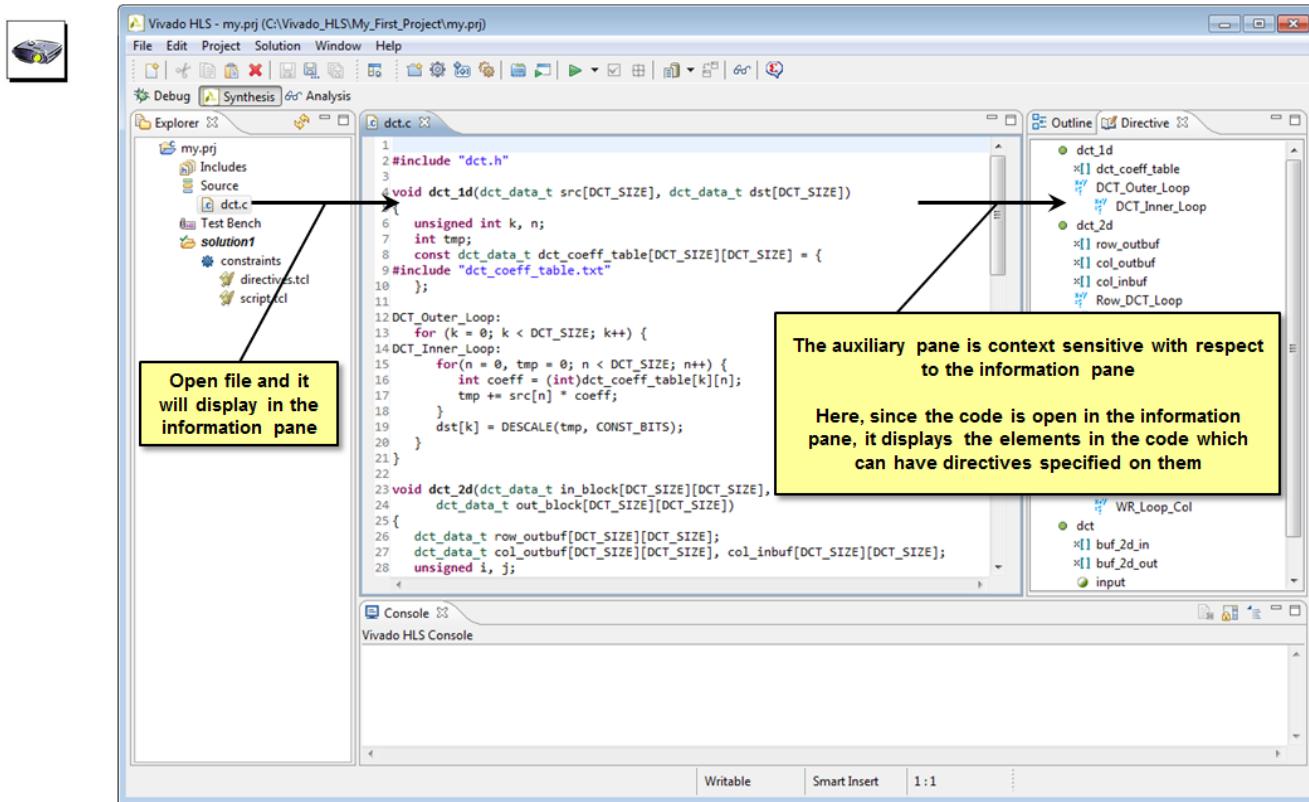


- Primary commands have toolbar buttons
 - Allows easy access for the most common functions
 - Run C simulation, run C synthesis and C/RTL co-simulation, and export RTL and open reports



Slide 3-19:

Files: Views, Edits, and Information



Slide 3-20:

C Validation to IP Creation

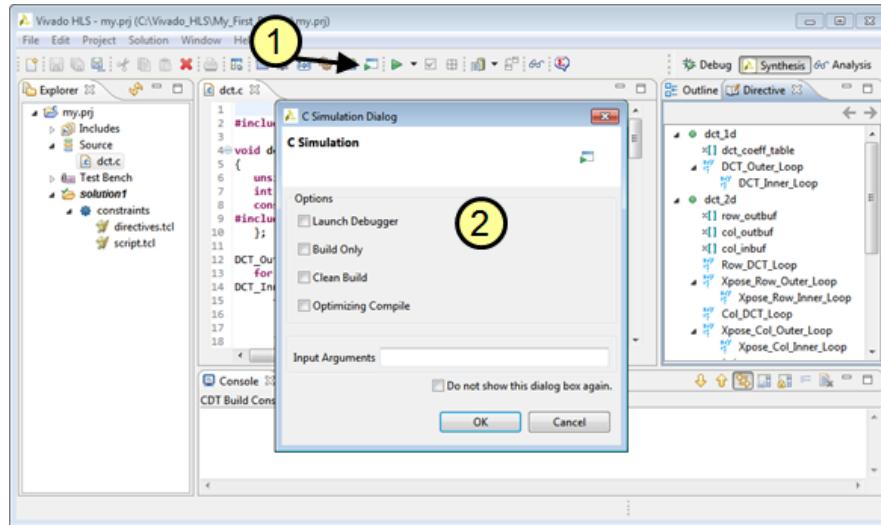
- Invoking the Vivado HLS Tool
- Projection Creation in the Vivado HLS Tool
- **C Validation to IP Creation**
- Adding Directives
- Vivado HLS Tool Directory Structure
- Summary

Slide 3-21:

Run C Simulation



1. Run C Simulation toolbar button
 - Single toolbar button to compile and execute C
2. C Simulation button opens a dialog box



- Default is to compile in debug mode and execute
 - Options modify behavior
3. Results
 - When run, the results will be shown in the console
 - Any messages output by the testbench are shown here

Command line: `csim_design`

Compiles and runs pre-synthesis provided C testbench

Slide 3-22:

C Simulation Options



Option	Behavior	Comments
Launch Debugger	Compile and then open the debugger	Incompatible with Optimizing Compile option
Build Only	Compile but do not execute	Compiles in debug mode if no other option selected
Clean Build	Clean up any existing files before starting	Can be used with all other options
Optimizing Compile	Compile in optimized (release) mode	No debug information is created; incompatible with using the debugger

Slide 3-23:

C Validation: Debugger



1. Controls

- Step (F5)
- Step-Over (F6)
- Step-Out (F7)

2. Breakpoint

- Double-click line number to create a breakpoint
- Run to next breakpoint (F8)

3. View values

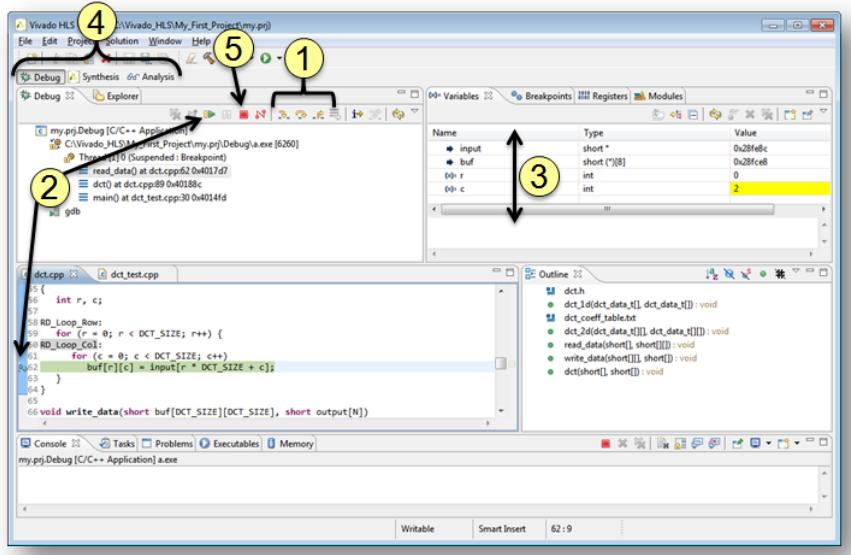
- Current values of C variables are shown

4. Perspectives

- Change between debug and synthesis perspectives

5. Terminate to finish

- To save memory resource, end the session when it is no longer required

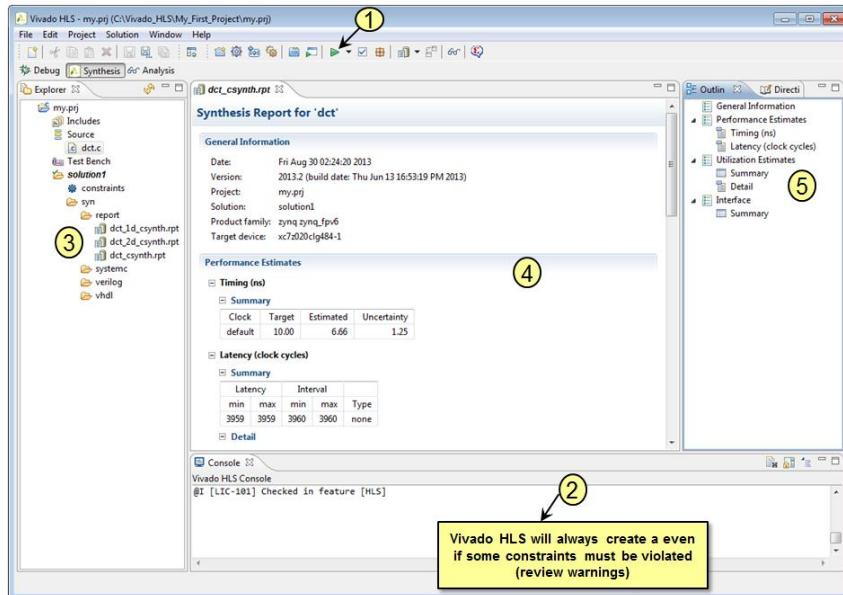


Slide 3-24:

Run C Synthesis



1. Execute synthesis
2. Console
 - Will show run-time information
 - Examine for failed constraints
3. *syn* directory is created
 - Verilog and VHDL RTL
 - Synthesis reports for all non-inlined functions
4. Report opens automatically
 - When synthesis completes
 - Opens in Report perspective
5. Report is outlined in the Auxiliary pane in the Synthesis perspective and the Explorer pane in Report perspective

Command line: `csynth_design`

Synthesizes the design for the active solution

Slide 3-25:

Analysis Perspective



- Perspective for design analysis
 - Allows interactive analysis

Module Hierarchy
Hierarchical Summary and Navigation

Performance Profile
Latency and Interval summary for this block

Schedule Viewer - dct

Current Module : dct

Operations\Control S: C0 C1 C2 C3 C4 C5

RD_Loop_Row (Yellow) | RD_Loop_Col (Green) | WR_Loop_Row (Yellow) | WR_Loop_Col (Green)

Performance View
Scheduled operations.

Loops: shown in Yellow are expandable and collapsible

Modules: shown in Green open the view on sub-blocks

Performance Resource Sharing

Slide 3-26:

Performance View



Hierarchical Navigation

Current Module : dct > dct 2d > dct 1d

Operations\Control S: C0 C1 C2 C3 C4 C5

i_21_read(wire read) | i_2_read(wire read) | [-]DCT_Outer_Loop | exitcond(icmp) | k_1(+) | [+]-DCT_Inner_| | tmp_1(+) | p_addr3(+) | node_60(write)

Loop Hierarchy

Scheduled States

Select operations and right-click to cross reference with the C source and HDL

C Source

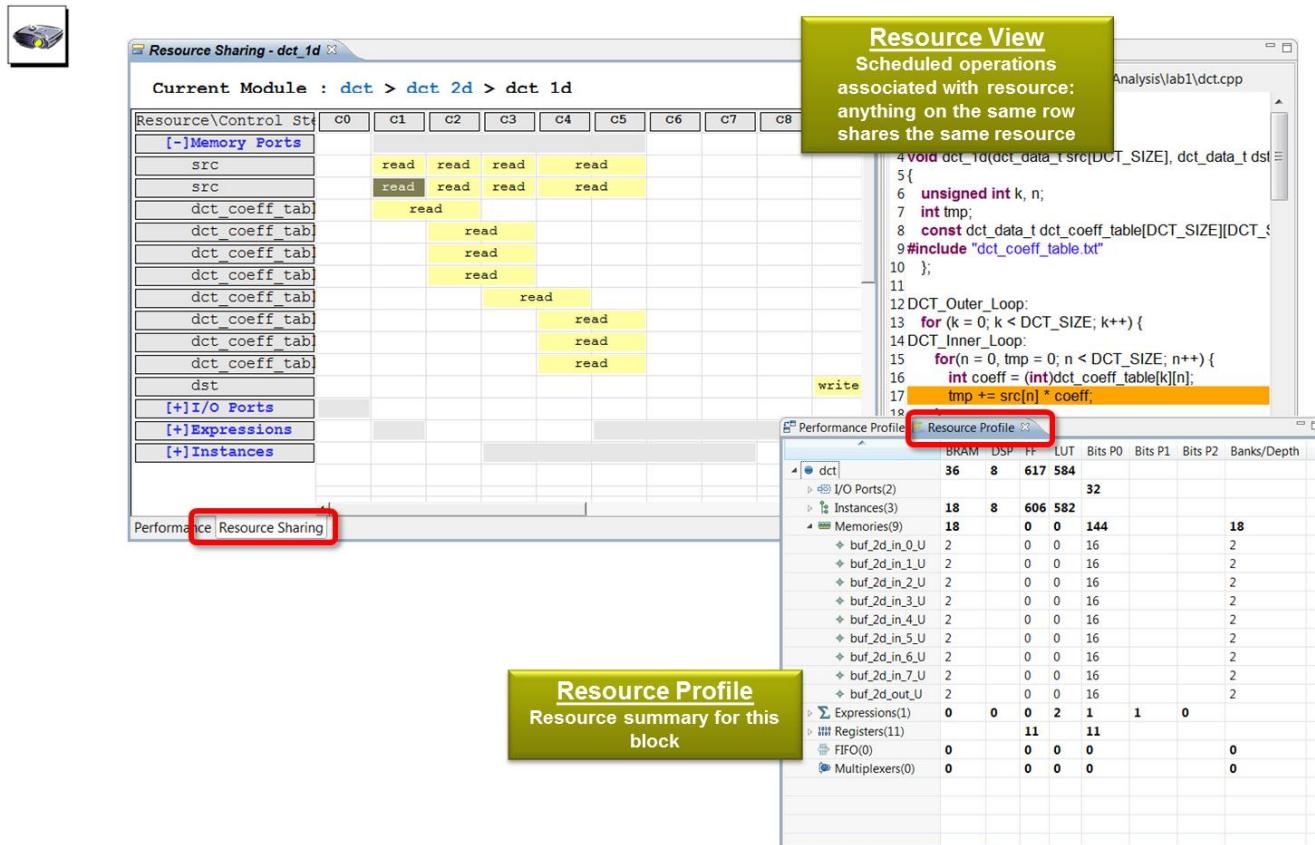
```

File: C:\Vivado_HLS_Tutorial\Design_Analysis\lab1\dct.c
1|
2#include "dct.h"
3|
4void dct_1d(dct_data_t src[DCT_SIZE], dct_data_t dst[DCT_SIZE]) {
5{
6    unsigned int k, n;
7    int tmp;
8    const dct_data_t dct_coeff_table[DCT_SIZE];
9#include "dct_coeff_table.txt"
10 };
11
12 DCT_Outer_Loop:
13     for (k = 0; k < DCT_SIZE; k++) {
14         for (n = 0; n < DCT_SIZE; n++) {
15             dst[k] = DESCALE(tmp, CONST_BITS);
16             tmp = src[n] * dct_coeff_table[k][n];
17         }
18     }
19 }
20
21
22
23 void dct_2d(dct_data_t in_block[DCT_SIZE][DCT_SIZE], dct_data_t out_block[DCT_SIZE][DCT_SIZE]) {
24     dct_1d(in_block[0], out_block[0]);
25     for (int i = 1; i < DCT_SIZE; i++) {
26         dct_1d(in_block[i], out_block[i]);
27     }
28 }

```

Slide 3-27:

Resource Analysis

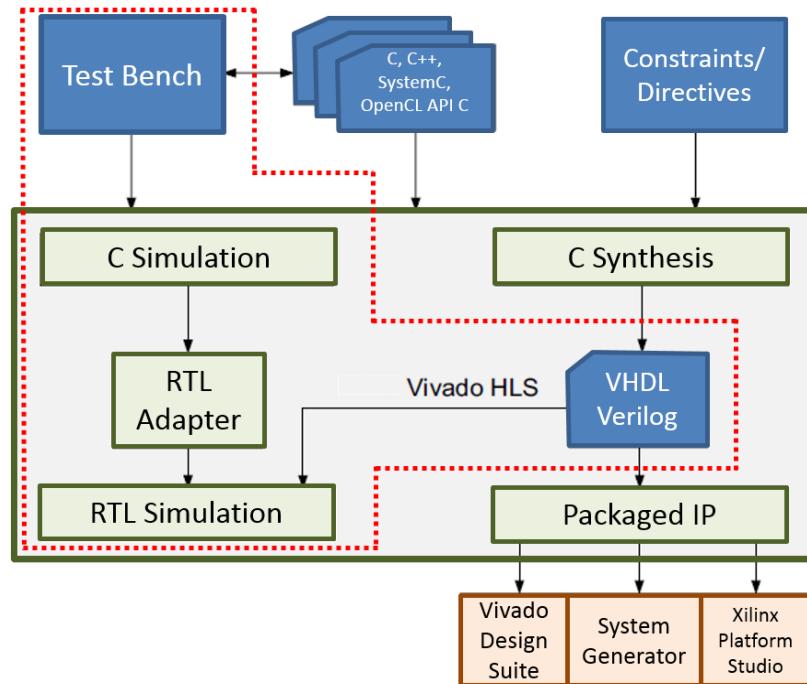


Slide 3-28:

Vivado HLS Tool: RTL Verification



- RTL output
 - Verilog
 - VHDL
- Automatic re-use of C-level testbench
- RTL verification executed within HLS
- Support for third-party HDL simulators

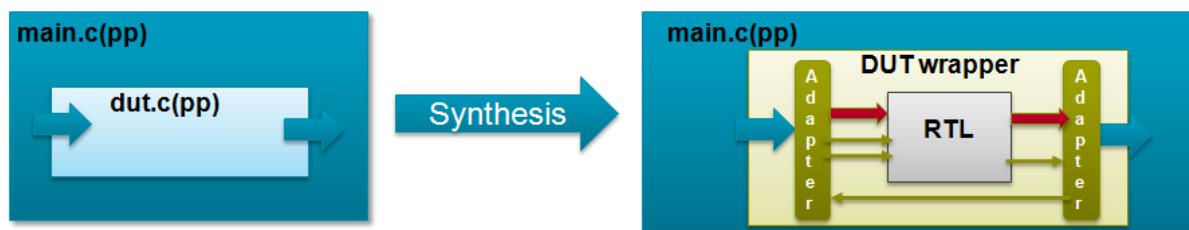


Slide 3-29:

RTL Verification: Under the Hood



- Co-simulation
 - Vivado HLS tool provides RTL verification
 - Creates the wrappers and adapters to re-use the C testbench



- Prior to synthesis
 - Testbench
 - Top-level C function
- After synthesis
 - Testbench
 - SystemC wrapper created by Vivado HLS tool
 - SystemC adapters created by Vivado HLS tool
 - RTL output from Vivado HLS tool
 - Verilog or VHDL

There is no HDL testbench created

Slide 3-30:

RTL Verification Support (1)



- Vivado HLS tool RTL output
 - Vivado HLS tool outputs RTL in Verilog, and VHDL
- Third-party HDL simulation support
 - Vivado HLS tool supports HDL simulators on both Windows and Linux
 - ***Third-party simulator executable must be in the Windows or Linux search path***
 - Requires a co-simulation license (HDL)

Slide 3-31:

RTL Verification Support (2)



Simulator	Linux	Windows
XSim (Vivado Simulator)	Supported	Supported
ISim (ISE Simulator)	Supported	Supported
Mentor Graphics ModelSim	Supported	Supported
Synopsys VCS	Supported	Not Available
Riviera	Supported	Supported



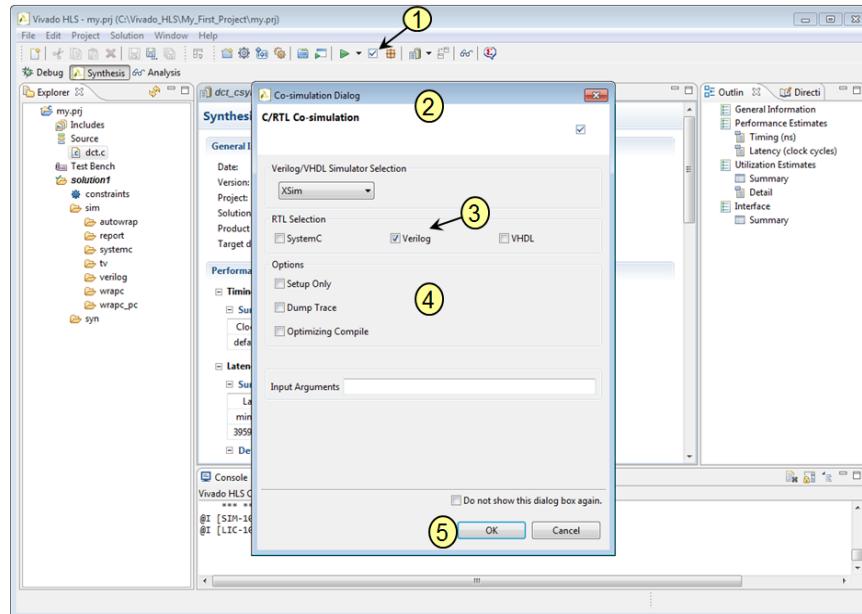
To verify an RTL design using third-party simulators, you must include the executable to the simulator in the system search path.

Slide 3-32:

Co-simulation



1. Start simulation
2. Opens the dialog box
3. Select the RTL
4. Options
 - Can output trace file (VCD format)
 - Optimize the C compilation and specify testbench linker flags



5. OK will run the simulator
6. Output files will be created in a *sim* directory

Command line: `cosim_design`

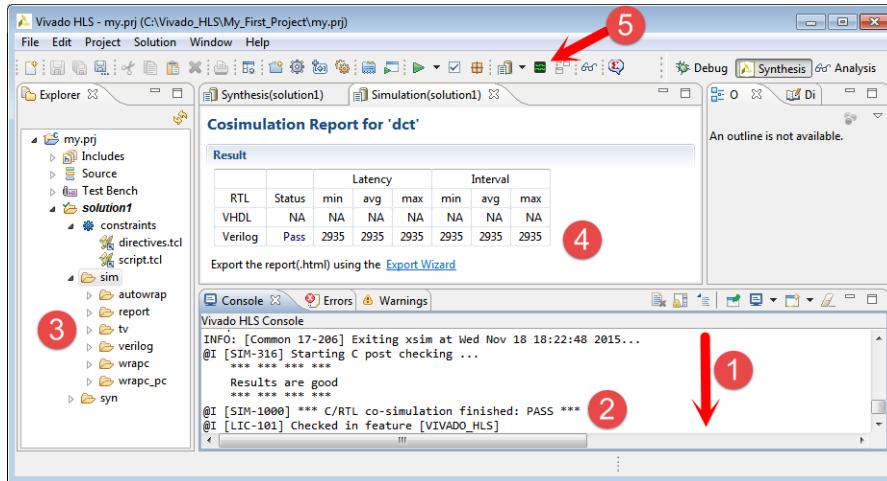
Executes the post-synthesis co-simulation of the synthesized RTL with the original C-based testbench

Slide 3-33:

Co-simulation Results



1. Simulation output is shown in the console
2. Expect the same testbench response
 - But slower
3. *sim* directory
 - Will contain a sub-directory for each RTL that is verified
4. Report
 - Report is created and opened automatically
5. Optional: waveform view
 - Opens the Waveform Viewer
 - Works only with the Vivado Simulator selection
 - RTL waveform opens in the Vivado Design Suite and allows analysis

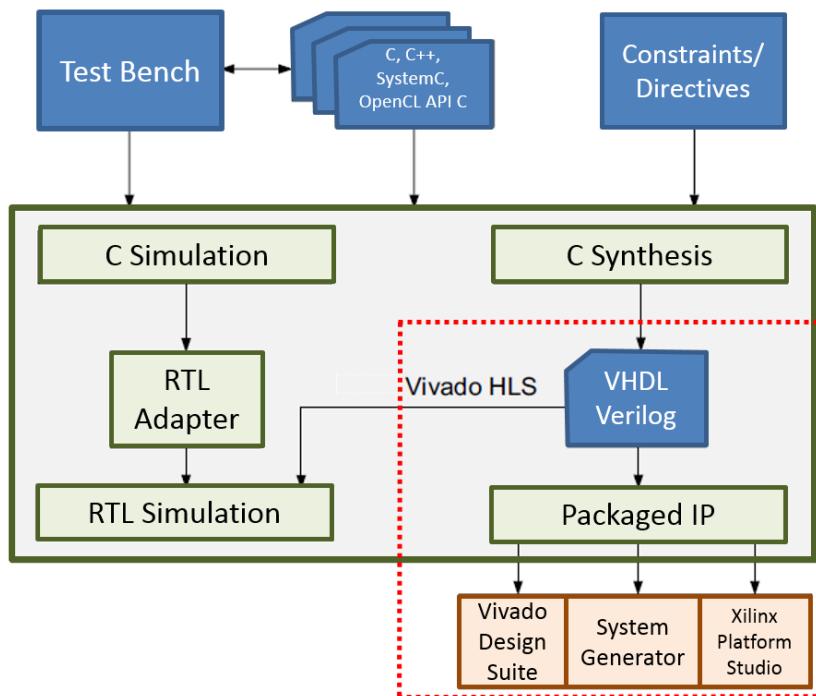


Slide 3-34:

Vivado HLS Tool: Exporting RTL as IP



- RTL output in
 - Verilog
 - VHDL
- RTL design exported as IP
 - Used in Xilinx and other tools
- RTL synthesis can be executed within HLS
 - For exploring results



Slide 3-35:

Export RTL Support (1)



- RTL synthesis
 - Allows RTL synthesis to be run from within the Vivado HLS tool
 - Supported using the Vivado or ISE tools
 - Select *Evaluate* option to perform RTL synthesis
 - Output files are written to *<solution name>/impl*
 - Top-level file to manually run synthesis is *impl.sh*

Slide 3-36:

Export RTL Support (2)



- RTL can be exported in the formats below
 - Vivado IP catalog format
 - 7 series devices and Zynq All Programmable SoC
 - Evaluate with Vivado synthesis
 - System Generator for DSP (Vivado synthesis) format
 - 7 series devices and Zynq All Programmable SoC
 - Evaluate with Vivado synthesis
 - System Generator for DSP (ISE) format
 - Virtex-6 and earlier devices
 - Evaluate with ISE synthesis
 - Requires VIVADO_HLS license
 - pcore for EDK format
 - Virtex-6 and earlier devices
 - Evaluate with ISE design tool synthesis
 - Requires VIVADO_HLS license



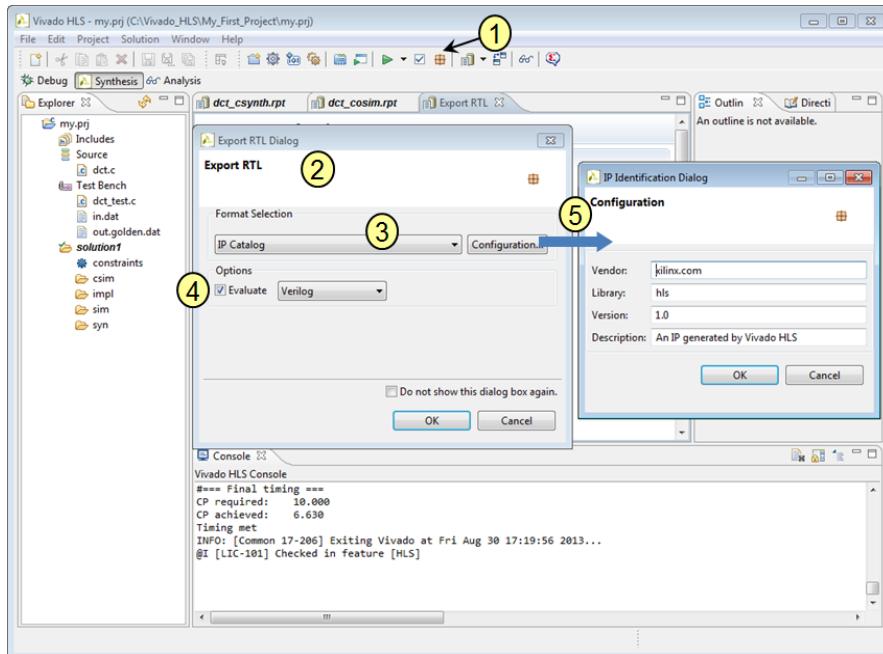
Reminder:

- HLS license come with Vivado System Edition
- VIVADO_HLS license is a standalone license with an upgrade
- Vivado HLS software is part of Vivado System Edition (no standalone install)

Slide 3-37:

Export RTL (RTL Synthesis)

1. Invoke **RTL Export**
2. Opens the dialog box
3. Select the format to export
4. Optionally evaluate the RTL by executing logic synthesis
5. Identification tags
 - Can be used to add metadata information to the IP block
6. Click **OK** to export (and optionally start synthesis)



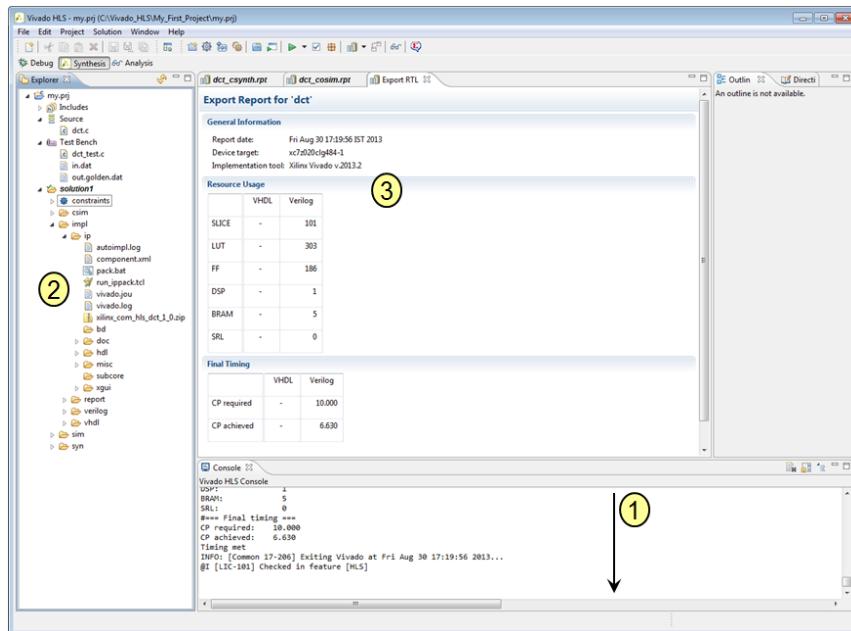
Command line: export_design	Exports and packages the synthesized design in RTL as an IP for downstream tools
--------------------------------	---

Slide 3-38:

Export RTL Results



1. Export RTL output is shown in the console
2. *impl* directory created
 - Will contain a sub-directory for each format for which it is exported
3. Report
 - Report is created and opened automatically



Slide 3-39:

Summary of Synthesis Flows (1)



Flow	Purpose	Output
Run C Simulation	Validate that the C algorithm has the correct functionality	Testbench should be self-checking
Run C Synthesis	Synthesize C to RTL	RTL output files
Multiple Solutions	Explore best area / performance trade-offs	RTL output files, user reports
C / RTL Co-simulation	Verify the RTL gives the same results as the C code using the same testbench (stimuli)	Testbench should be self-checking
Export RTL	Export the RTL as IP / System Generator block/pcore to use with other Xilinx tools	IP package (optionally the results of RTL synthesis for your review)

Slide 3-40:

Summary of Synthesis Flows (2)



- Vivado HLS tool will create results in an RTL synthesis directory
 - These results confirm that RTL synthesis matches those estimated by Vivado HLS tool
- Export RTL output in the *impl* directory should be used for final RTL synthesis
 - Take this RTL IP and combine it with the other blocks in RTL project / IPI project / System Generator
 - Then, everything is synthesized, placed, routed, and bitstream generated

Slide 3-41:

Adding Directives



- Invoking the Vivado HLS Tool
- Projection Creation in the Vivado HLS Tool
- C Validation to IP Creation
- **Adding Directives**
- Vivado HLS Tool Directory Structure
- Summary

Slide 3-42:

Applying Directives (1)

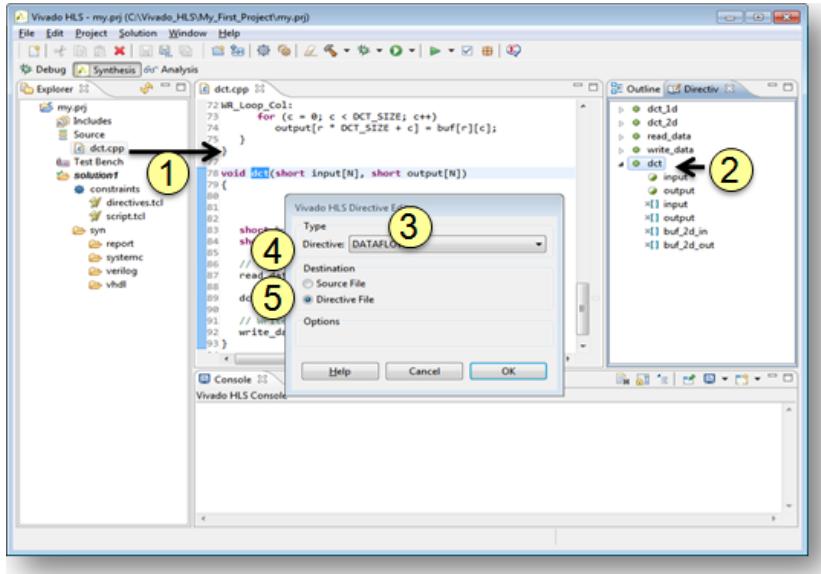


- If the source code is open in the GUI Information pane
 - The Directives tab in the Auxiliary pane shows all of the locations and objects upon which directives can be applied (in this CPP file, not the whole design)
 - Functions, loops, regions, arrays, top-level arguments

Slide 3-43:

Applying Directives (2)

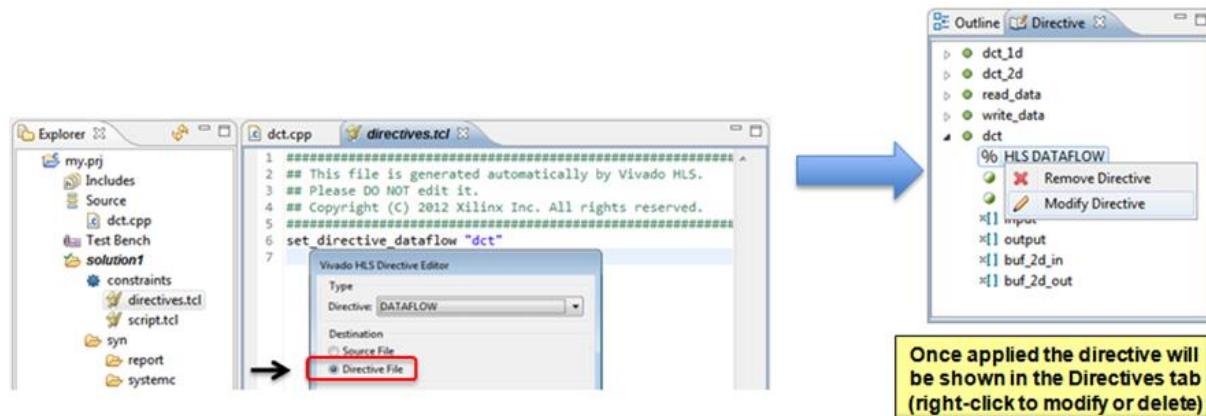
-  1. Open the source code
- 2. Select the object in the Directives Tab
 - This example selects a function
- 3. Right-click to open the dialog box
- 4. Select a directive from the drop-down menu
 - This example selects dataflow
- 5. Specify the destination
 - Discussed next
 - And any options



Slide 3-44:

Optimization Directives: Tcl

-  ■ Directives can be placed in the directives file
 - Tcl command is written into *directives.tcl*



Once applied, the directive will be shown in the Directives tab (right-click to modify or delete)

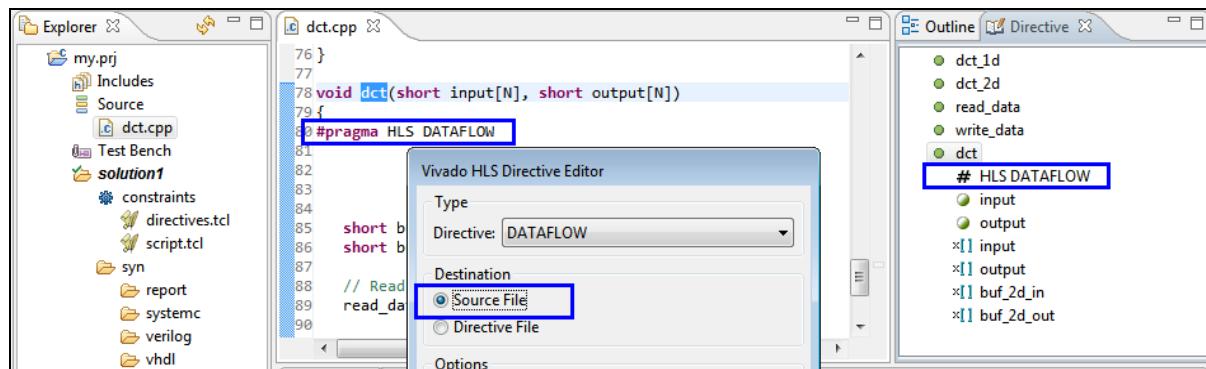
- A *directives.tcl* file in each solution
 - Each solution can have different directives

Slide 3-45:

Optimization Directives: Pragma



- Directives can be placed into the C source
 - Pragmas are added (and will remain) in the C source file
 - Pragmas will be used by every solution that uses the code

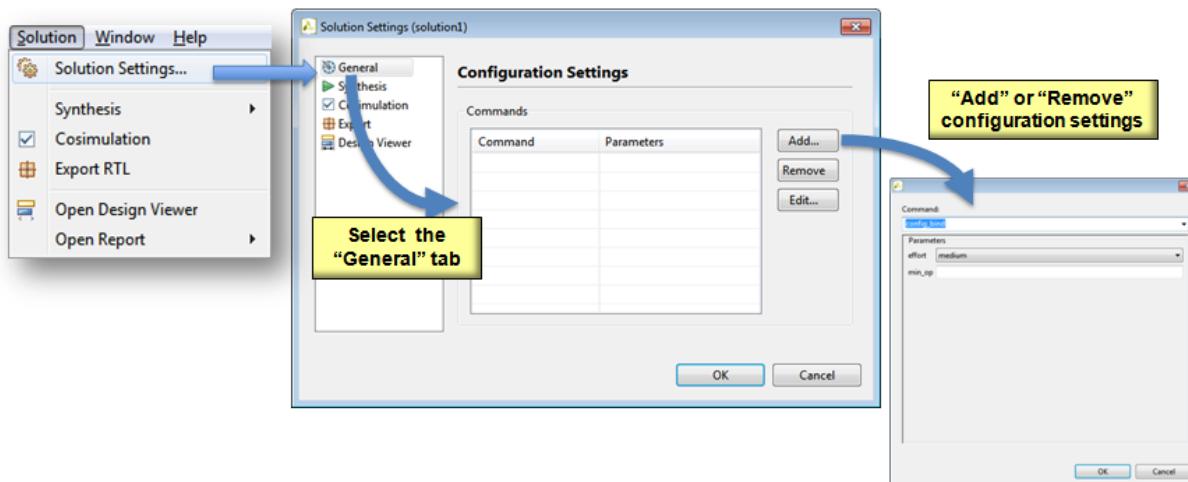


Slide 3-46:

Solution Configurations



- Configurations can be set on a solution
 - Set the default behavior for that solution
 - Open configurations settings from the menu (**Solutions > Solution Settings**)



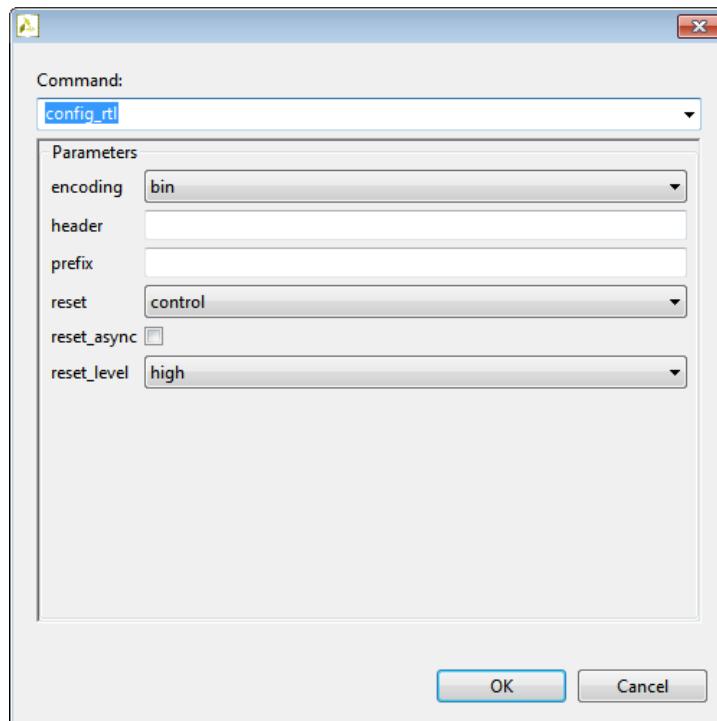
- Choose the configuration from the drop-down menu
 - Array Partitioning, Dataflow Memory types, Default I/O ports, RTL Settings, Operator binding, Schedule efforts

Slide 3-47:

Example: Configuring the RTL Output (1)



- Specify the FSM encoding style
 - By default the FSM is binary
 - Can also be one-hot or gray encoding
- Add a header string to all RTL output files
 - Example: *Copyright Acme Inc.*
- Add a user-specified prefix to all RTL output filenames
 - RTL has the same name as the C functions
 - Plus any RTL macro blocks created
 - Allow multiple RTL variants of the same top-level function to be used together without renaming files

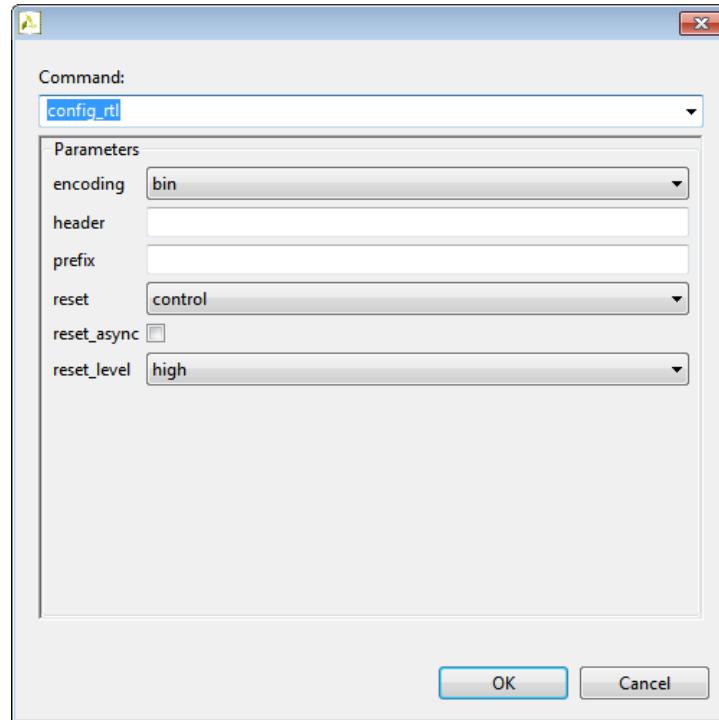


Slide 3-48:

Example: Configuring the RTL Output (2)



- Reset options
 - All variables initialized in the code are reset to their initial value
 - None: reset no other register in the RTL
 - Control (default): reset registers in the FSM and driving protocol handshakes signals
 - State: as Control plus those derived from statics/globals in the code
 - All: reset all registers



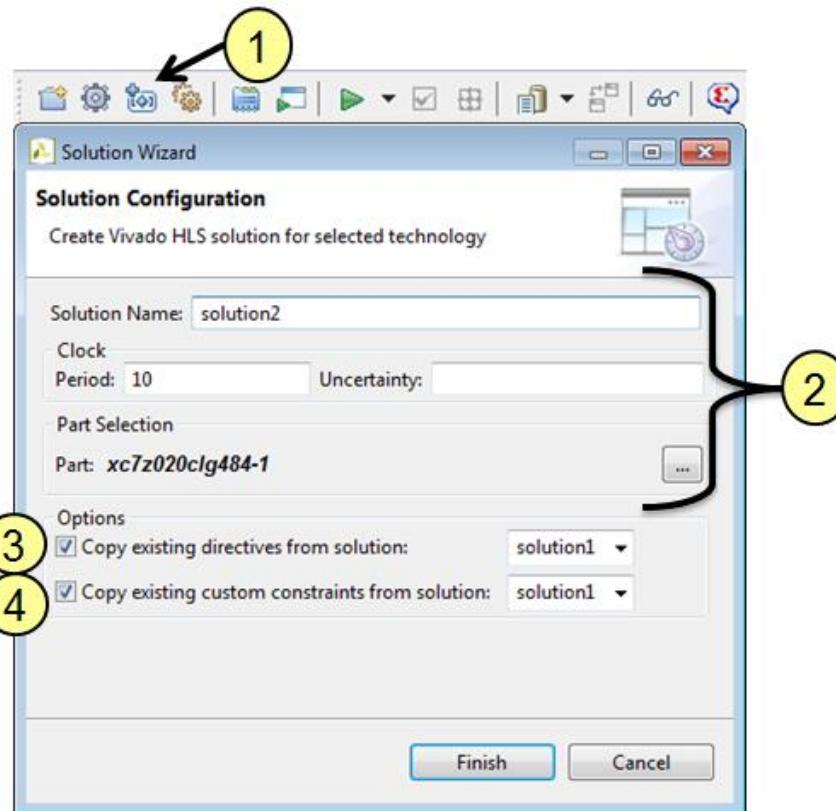
- Synchronous or asynchronous reset
 - Default is synchronous reset
- Active high or low reset
 - Default is active high

Slide 3-49:

Copying Directives into New Solutions



- When creating new solutions
- 1. Click the **New Solution** icon
- 2. Modify any of the settings
- 3. Copy existing directives
 - Or not
 - No need to copy pragmas, they are in the code
- 4. Copy any existing custom constraints in the existing *script.tcl* to the new *script.tcl*
 - Or not



Slide 3-50:

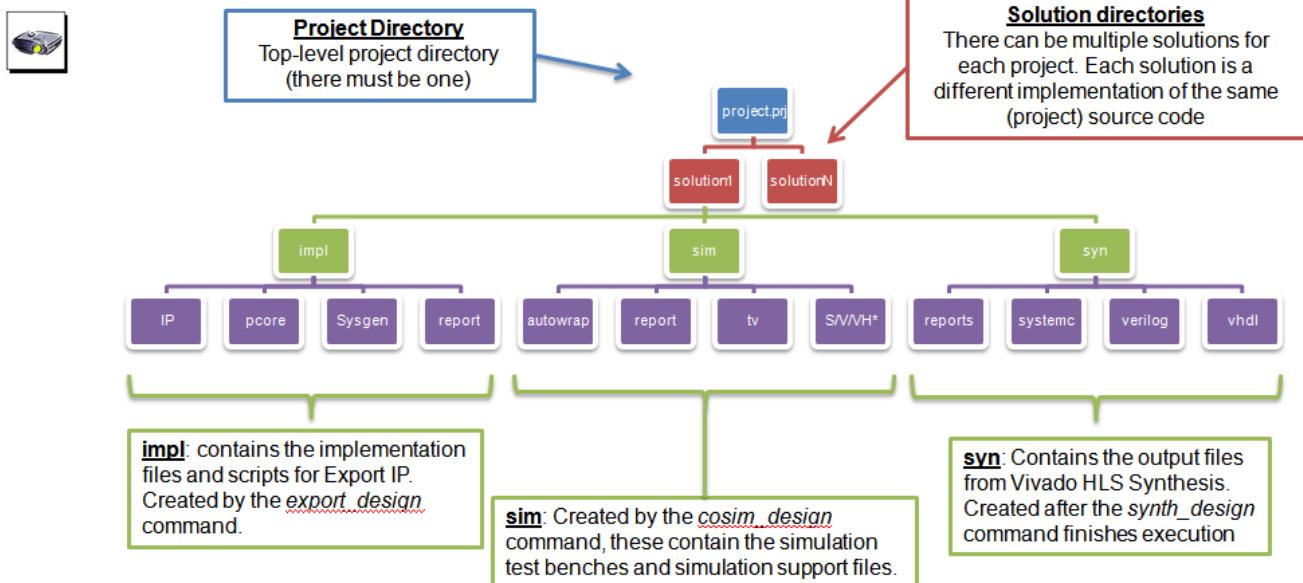
Vivado HLS Tool Directory Structure



- Invoking the Vivado HLS Tool
- Project Creation in the Vivado HLS Tool
- C Validation to IP Creation
- Adding Directives
- **Vivado HLS Tool Directory Structure**
- Summary

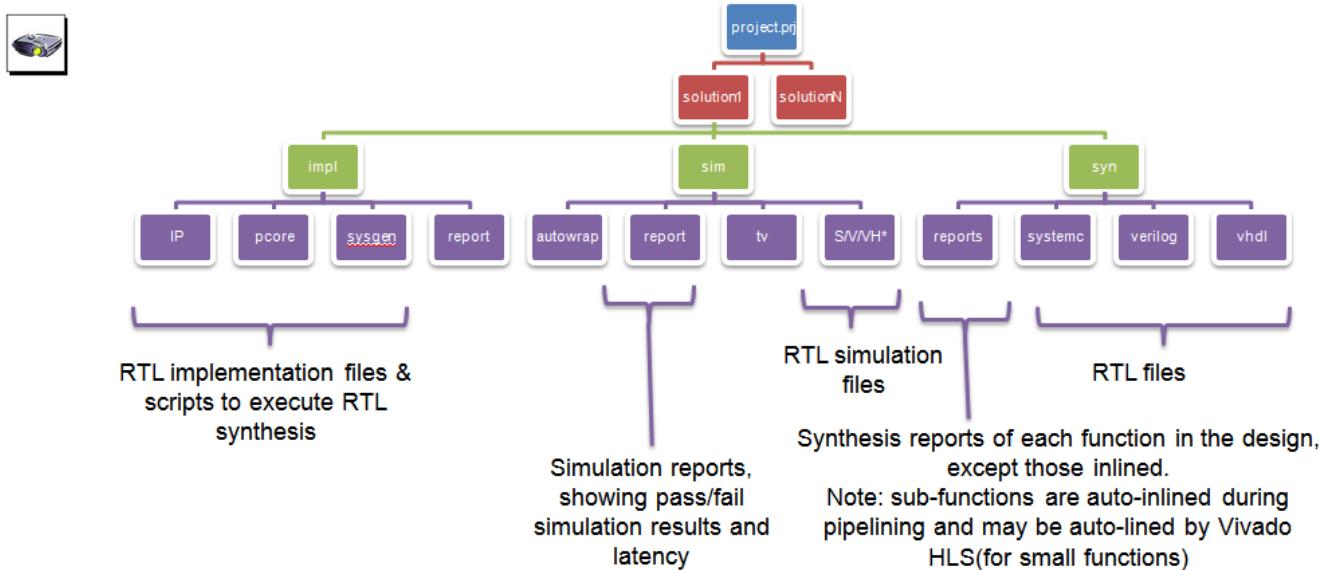
Slide 3-51:

Vivado HLS Tool Directory Structure (1)



Slide 3-52:

Vivado HLS Tool Directory Structure (2)



Slide 3-53:

Summary

- Invoking the Vivado HLS Tool
 - Projection Creation in the Vivado HLS Tool
 - C Validation to IP Creation
 - Adding Directives
 - Vivado HLS Tool Directory Structure
 - **Summary**
- 

Slide 3-54:

Summary (1)



- Vivado HLS tool project creation wizard involves
 - Defining project name and location
 - Adding design files
 - Specifying testbench files
 - Selecting clock and technology
- Vivado HLS tool can be used from the GUI or command line interface mode
- Top-level module in testbench is *main()*, whereas the top-level module in the design is the function to be synthesized
- Constraints
 - Directives or pragmas are the way to override the defaults
 - Directives will be saved in the associated *directives.tcl* file
 - Pragmas will reside in the source code

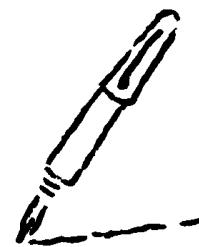
Slide 3-55:

Summary (2)



- Vivado HLS tool project directory consists of
 - *.prj project file
 - Multiple solutions directories
 - Each solution directory may contain
 - *impl*, *synth*, and *sim* directories
 - *impl* directory consists of IP Export, Verilog, and VHDL folders
 - *synth* directory consists of reports, SystemC, VHDL, and Verilog folders
 - *sim* directory consists of testbench and simulation files
- Multiple solutions can be configured in the Vivado HLS tool
 - Typically, each solution with different directives and/or different solution settings

Capture Your Notes Here



I/O Interfaces

Slide 4-1:

2016.1

This module describes the interfaces abstracted by the Vivado® HLS tool from the C design. 75 minutes.

Slide 4-2:

Objectives

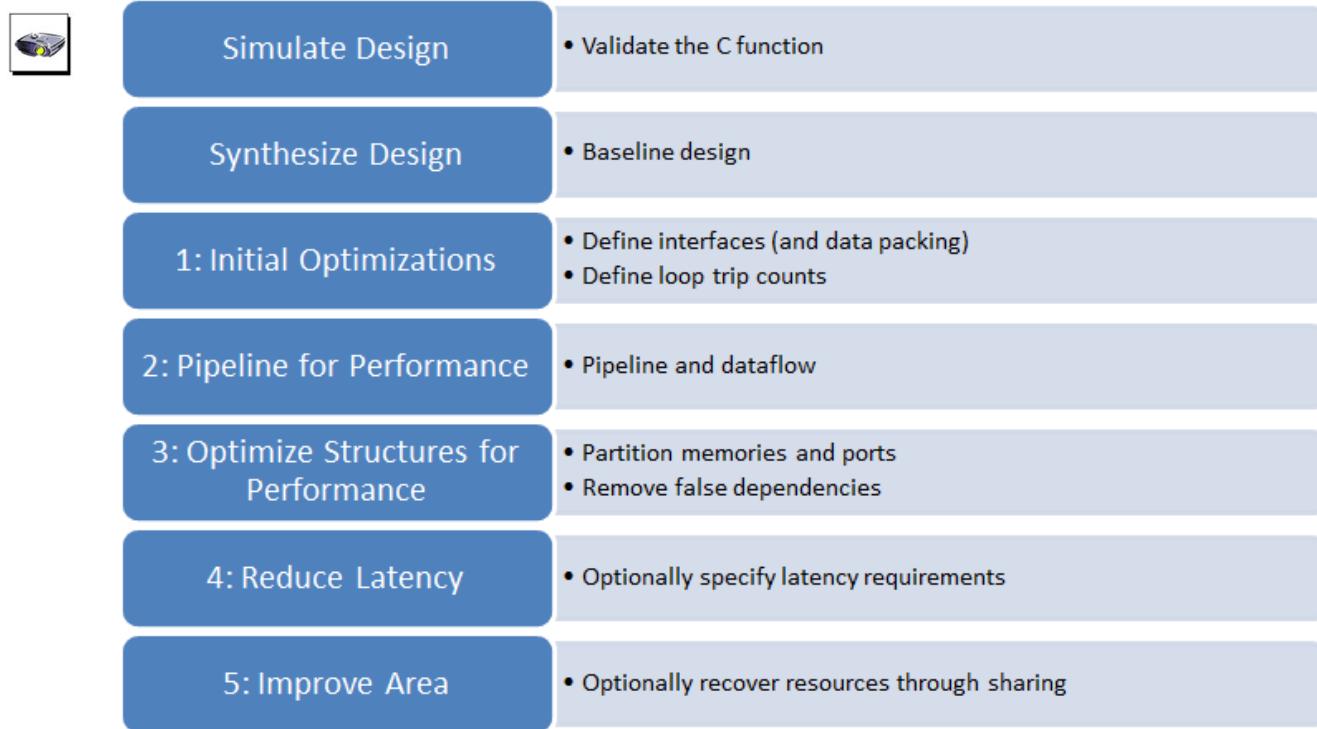


After completing this module, you will be able to:

- List the types of I/O abstracted in the Vivado HLS tool
- List various basic and optional I/O ports handled in the Vivado HLS tool
- Distinguish between block-level and port-level I/O protocols
- State how pointer interfaces are implemented

Slide 4-3:

HLS UltraFast Design Methodology



The design interface is typically defined by the other blocks in the system. Since the type of I/O protocol helps determine what can be achieved by synthesis it is recommended that the INTERFACE directive be used to specify this before proceeding to optimize the design.

If the algorithm accesses data in a streaming manner, you might want to consider using one of the streaming protocols to ensure high-performance operation.

Slide 4-4:

Step 1: Initial Optimizations – Directives



Directives and Configurations	Description
INTERFACE	Specifies how RTL ports are created from the function description.
DATA_PACK	Packs the data fields of a struct into a single scalar with a wider word width.
LOOP_TRIPCOUNT	Used for loops that have variable bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.
Config Interface	This configuration controls I/O ports not associated with the top-level function arguments and allows unused ports to be eliminated from the final RTL.



The design interface is typically defined by the other blocks in the system. Since the type of I/O protocol helps determine what can be achieved by synthesis, it is recommended that the INTERFACE directive be used to specify this before proceeding to optimize the design.

If the algorithm accesses data in a streaming manner, you might want to consider using one of the streaming protocols to ensure high performance operation.

Tip: If the I/O protocol is completely fixed by the external blocks and will never change, consider inserting the INTERFACE directives directly into the C code as pragmas.

When structs are used in the top-level argument list, they are decomposed into separate elements and each element of the struct is implemented as a separate port. In some cases, it is useful to use the DATA_PACK optimization to implement the entire struct as a single data word, resulting in a single RTL port.

Care should be taken if the struct contains large arrays. Each element of the array is implemented in the data word and this might result in a very wide data ports.

A common issue when designs are first synthesized is report files showing the latency and interval as a question mark "?" rather than as numerical values. If the design has loops with variable loop bounds, the Vivado® HLS tool cannot determine the latency and uses the "?" to indicate this condition.

To resolve this condition, use the analysis perspective or the synthesis report to locate the lowest level loop for which synthesis fails to report a numerical value and use the LOOP_TRIPCOUNT directive to apply an estimated tripcount. This allows values for latency and interval to be reported and allows solutions with different optimizations to be compared.

Finally, global variables are generally written to and read from, within the scope of the function for synthesis and are not required to be I/O ports in the final RTL design. If a global variable is used to bring information into or out of the C function, you might want to expose them as an I/O port using the interface configuration.

Slide 4-5:

Overview of Vivado HLS Tool I/O Ports, Protocols, and Options



- **Overview of Vivado HLS Tool I/O Ports and Protocols**
 - Block-Level I/O Protocols
 - Port-Level I/O Protocols
 - DATA_PACK Directive
 - Summary

Slide 4-6:

Key Attributes of C Code: I/O



- **Functions:** All code is made up of functions that represent the design hierarchy; the same in hardware

Input & Outputs: The arguments of the top-level function must be transformed to hardware interfaces with an IO protocol

- **Types:** All variables are of a defined type. The type can influence the area and performance
- **Loops:** Functions typically contain loops. How these are handled can have an impact on area and performance
- **Arrays:** Arrays are used often in C code. They can impact the device area and become performance bottlenecks
- **Operators:** Operators in the C code may require sharing to control area or to be assigned to specific hardware implementations to meet performance

```
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>>2;  
}
```

Slide 4-7:

Example 101: Combinatorial Design



- Simple adder example
 - Output is the sum of three inputs

```
#include "adders.h"
int adders(int in1, int in2, int in3) {
    int sum;
    sum = in1 + in2 + in3;
    return sum;
}
```

- Synthesized with 100-ns clock
 - All adders can fit in one clock cycle
 - **Combinatorial design**



```
=====
== Performance Estimates
=====
+ Summary of timing analysis:
  * Estimated clock period (ns): 3.45
+ Summary of overall latency (clock cycles):
  * Best-case latency: 0
  * Average-case latency: 0
  * Worst-case latency: 0
```

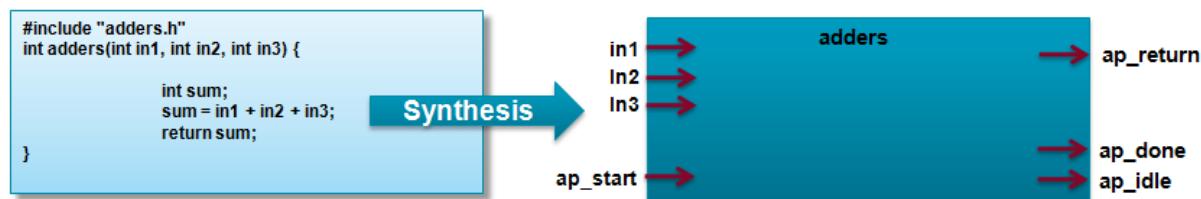
- Function return becomes RTL port ap_return
 - Vivado HLS tool port (ap_) is used in all ports added by the Vivado HLS tool
- No handshakes are required or created in this example

Slide 4-8:

Example 102: Sequential Design



- Same adder design is now synthesized with a 3-ns clock
 - Design now takes more than one cycle to complete
 - Vivado HLS tool creates a **sequential** design with the default port types



- By default
 - Block-level handshake signals are added to the design
 - These tell the design when to start operation (ap_start)
 - Indicate when the design has completed (ap_done) and is idle (ap_idle)
 - In a pipelined design, they also indicate when the design can accept new data (ap_ready): covered later

Slide 4-9:

Type of I/O Protocol



- Vivado HLS tool supports two solutions for specifying the type of I/O protocol
 - Interface synthesis
 - Where the port interface is created based on efficient industry-standard interfaces
 - Manual interface specification
 - Where the behavior is explicitly described in the source code



Manual interface specification is where the interface behavior is explicitly described in the input source code. This allows any arbitrary I/O protocol to be used.

This solution is provided through SystemC designs, where the I/O control signals are specified in the interface declaration and their behavior is specified in the code.

The Vivado HLS tool also supports this mode of interface specification for C and C++ designs.

Slide 4-10:

Interface: Block Level and Port Level



- Interface type is determined by the **interface** directive
- Two categories of interface
 - Block-level interface protocol
 - Controls the RTL block
 - Port-level interface protocol
 - Sends the data into and out of the block



Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream	
	Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none										
ap_ctrl_hs		D								
ap_ctrl_chain										
axis										
s_axilite										
m_axi										
ap_none	D						D			
ap_stable										
ap_ack										
ap_vld								D		
ap_ovld								D		
ap_hs										
ap_memory			D	D	D					
bram										
ap_fifo										D
ap_bus										

Supported D = Default Interface

Not Supported



Block-Level Interface Protocol

By default, a block-level interface protocol is added to the design. These signals control the block, independently of any port-level I/O protocols. These ports control when the block can start processing data (`ap_start`), indicate when it is ready to accept new inputs (`ap_ready`), and indicate if the design is idle (`ap_idle`) or has completed operation (`ap_done`).

Port-Level Interface Protocol

The final group of signals are the data ports. The I/O protocol created depends on the type of C argument and on the default. A complete list of all possible I/O protocols is shown in the table.

After the block-level protocol has been used to start the operation of the block, the port-level I/O protocols are used to sequence data into and out of the block.

Slide 4-11:

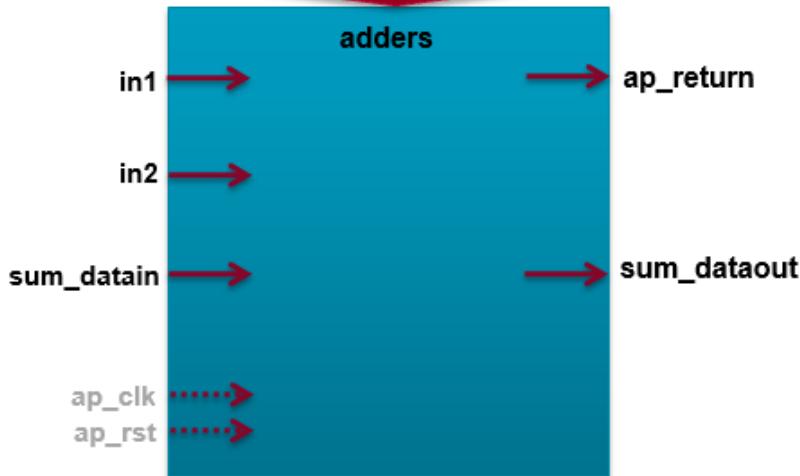
Vivado HLS Optional I/O: Function Arguments



- Function arguments
 - Synthesized into data ports
- Function return
 - Any return is synthesized into an output port called `ap_return`
- Pointers (and C++ references)
 - Can be read from and written to
 - Separate input and output ports for pointer reads and writes

```
#include "adders.h"
int adders(int in1, int in2, int *sum) {
    int temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;
    return temp;
}
```

Synthesis



- Arrays (not shown here)
 - Like pointers can be synthesized into read and/or write ports

Slide 4-12:

Vivado HLS Optional I/O: Block-Level Protocol



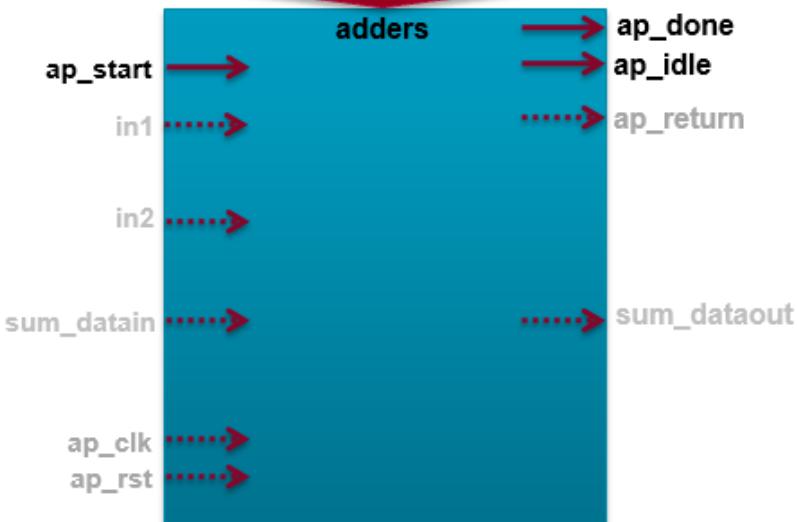
- Block-level protocol
 - An I/O protocol added at the RTL block level
 - Controls and indicates the operational status of the block
- Block operation control
 - Controls when the RTL block starts execution (`ap_start`)
 - Indicates if the RTL block is idle (`ap_idle`) or has completed (`ap_done`)
- Complete and function return
 - `ap_done` signal also indicates when any function return is valid
- Ready (not shown here)
 - If the function is pipelined an additional ready signal (`ap_ready`) is added
 - Indicates a new sample can be supplied before done is asserted

```
#include "adders.h"
int adders(int in1, int in2, int *sum) {

    int temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;

    return temp;
}
```

Synthesis



Slide 4-13:

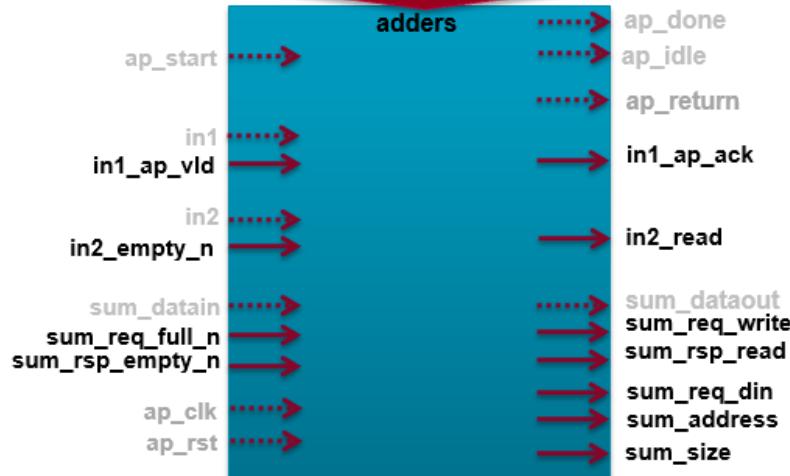
Vivado HLS Optional I/O: Port-Level I/O Protocols



- Port I/O protocols
 - I/O protocol added at the port level
 - Sequences the data to/from the data port

```
#include "adders.h"
int adders(int in1, int in2, int *sum) {
    int temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;
    return temp;
}
```

Synthesis



- Interface synthesis
 - Design is automatically synthesized to account for I/O signals
- Select from a predefined list
 - I/O protocol for each port can be selected from a list
 - Allows the user to easily connect to surrounding blocks
- Non-standard interfaces
 - Supported in C/C++ using an arbitrary protocol definition
 - Supported natively in SystemC

Slide 4-14:

Interface Directive



- Block-level I/O protocol (yellow) is specified by using the interface directive with `port=return`
- Port-level I/O protocol (green) is specified by using the interface directive with `port=<port name>`

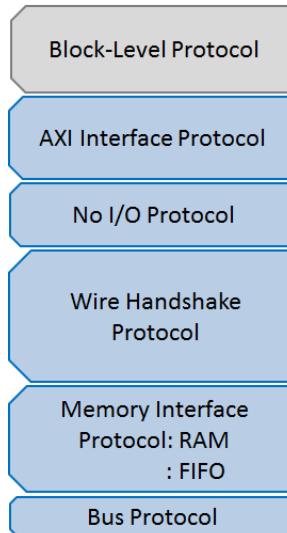
```
4
5  out_T dot(in_T a[N], in_T b[N], in_T init, out_T *total )  {
6
7      #pragma HLS INTERFACE s_axilite port=return
8
9      #pragma HLS INTERFACE ap_fifo depth=16 port=b
10     #pragma HLS INTERFACE ap_fifo depth=16 port=a
11
12
13     int sum = init;
14     int tmp = 0;
15
16     for(int i=0; i < N; i++){
17         tmp = a[i] * b[i];
18         sum += tmp;
19     }
20
21     *total = *total + sum;
22     return sum;
23
24 }
25
```

Slide 4-15:

Port Level I/O



- Port-level I/O interfaces are created for each argument in the top-level function (and the function return)
- Three types of function arguments
 - Scalar
 - Array
 - Pointer or reference



Argument Type	Scalar		Array		Pointer or Reference		HLS Stream		
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld								D	
ap_ovld							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo									
ap_bus									D

Supported D = Default Interface Not Supported

- Each type has a default interface type

Slide 4-16:

Vivado HLS Tool Interfaces



- Summary can be found in the Synthesis report

The screenshot shows the Vivado HLS Synthesis report for the 'adders.rpt' file. The 'Interface Summary' section is highlighted with a yellow box. It contains tables for 'Interfaces' and 'Ports'. The 'Interfaces' table lists objects like ap_clk, ap_rst, ap_start, ap_done, in1, in1_ap_vld, in1_ap_ack, in2_dout, in2_empty_n, in2_read, sum_req_din, sum_req_full_n, sum_req_write, sum_rsp_dout, sum_rsp_empty_n, sum_rsp_read, sum_address, sum_datain, sum_dataout, and sum_size, along with their respective types, scopes, IO protocols, IO configurations, directions, and bit widths. The 'Ports' table lists ports like in1, in1_ap_vld, in1_ap_ack, in2_dout, in2_empty_n, in2_read, sum_req_din, sum_req_full_n, sum_req_write, sum_rsp_dout, sum_rsp_empty_n, sum_rsp_read, sum_address, sum_datain, sum_dataout, and sum_size, with their types, scopes, IO protocols, IO configurations, directions, and bit widths. A large yellow box labeled 'Interface Summary section (at the end of the report)' covers the right side of the interface summary table, which includes sections like Report Version, General Information, User Assignments, Performance Estimates, Area Estimates, Power Estimate, and Interface Summary.

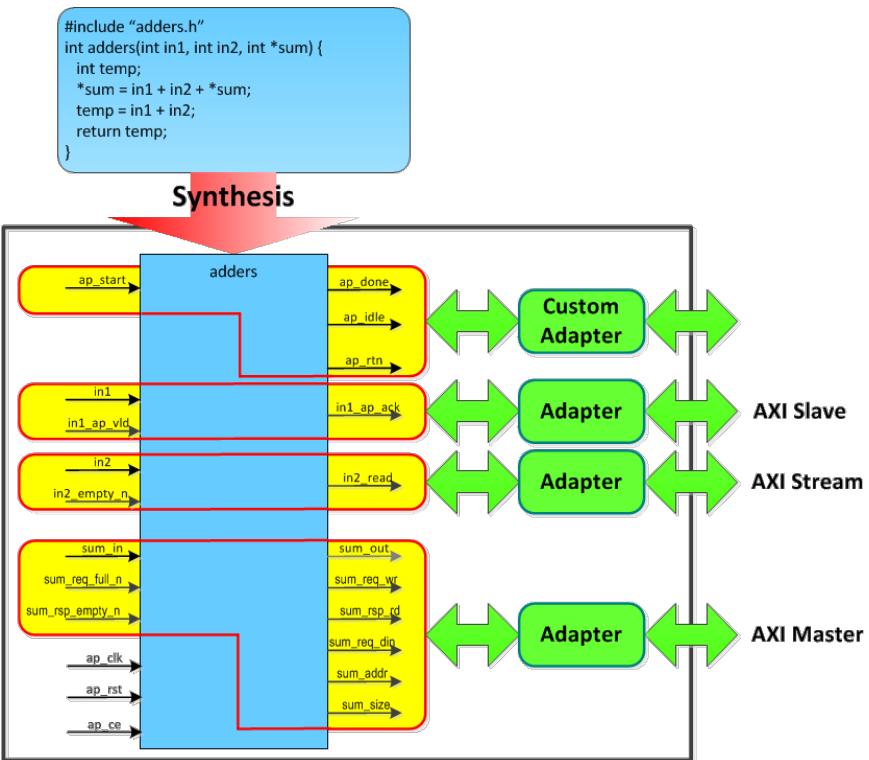
Slide 4-17:

Vivado HLS Tool I/O Options: IP Adapters



- Bus interfaces
 - For use in Vivado IP integrator environment

- Bus protocols
 - AXI4-Stream
 - AXI4-Master
 - AXI4-Slave



Slide 4-18:

Block-Level I/O Protocols



- Overview of Vivado HLS Tool I/O Ports and Protocols
- **Block-Level I/O Protocols**
- Port-Level I/O Protocols
- DATA_PACK Directive
- Summary

Slide 4-19:

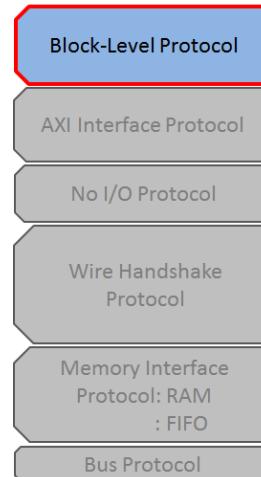
Block-Level I/O



- Block-level I/O controls the HLS module behavior
 - Start the module
 - Monitor the status
 - idle/busy (ap_idle), ready for new input (ap_ready), done (ap_done)

- Three types of block-level I/O
 - ap_ctrl_none
 - ap_ctrl_hs
 - ap_ctrl_chain

- s_axi_lite
 - Use AXI-Lite to memory-map all the block-level I/O
 - Processor can control the block



Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
	Input	Return	I	I/O	O	I	I/O	O	I and O
Interface Mode									
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld							D		
ap_ovald								D	
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo								D	
ap_bus									

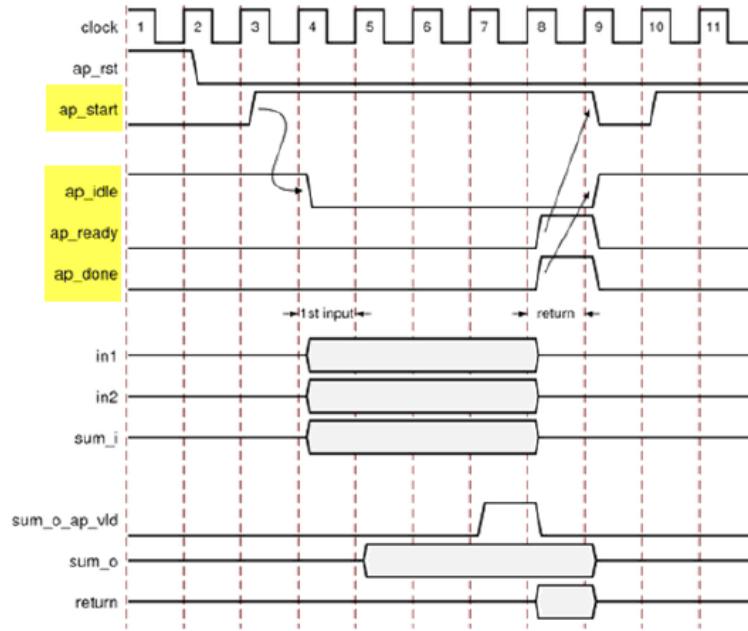
■ Supported D = Default Interface ■ Not Supported

Slide 4-20:

Block Level I/O: ap_ctrl_hs



- ap_ctrl_hs: Default block-level I/O
 - Design starts when ap_start is asserted High
 - ap_idle is asserted Low to indicate the design is operating
 - Input data is read at any clock after the first cycle. Vivado HLS schedules when the reads occur.
ap_ready is asserted High when all inputs have been read
 - When output sum is calculated, the associated output handshake (sum_o_ap_vld) indicated 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 to start again

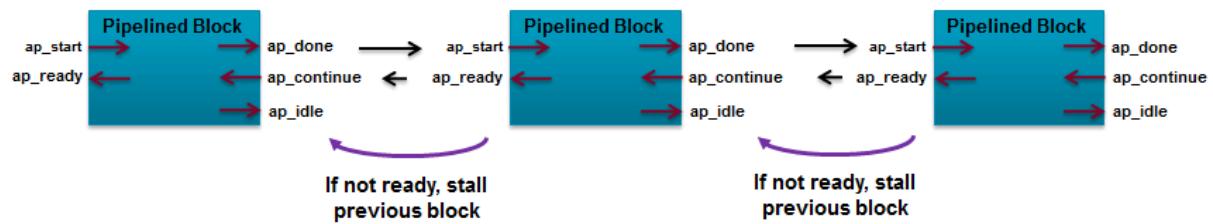


Slide 4-21:

Block Level I/O Protocol: ap_ctrl_chain



- Protocol to support pipeline chains: ap_ctrl_chain
 - Similar to ap_ctrl_hs, except ap_continue input signal
 - Allows blocks to be easily chained in a pipelined manner
 - ap_ctrl_chain protocol provides back-pressure in systems
 - Useful when integrating several HLS modules together in RTL domain



Slide 4-22:

Block Level I/O: ap_ctrl_none



- No block-level IO
 - No ap_start, ap_idle, ap_ready, ap_done
- Useful for module where there is no need to control stop/go
- For example: Combinatorial logic, "**I=1**" module
- Sometimes this can affect **I** in the synthesis

Slide 4-23:

Block Level I/O: s_axilite

- AXI4-lite interface can be used to control the HLS block by memory-mapping the block level I/O (ap_start, ap_idle, ap_done, ap_ready)
- A common practice when the processing system (PS)/processor is used to configure the HLS block
- Shown in the header file generated by HLS when s_axilite is used for the block-level I/O protocol

```
// 0x00 : Control signals
//   bit 0 - ap_start (Read/Write/SC)
//   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)
// 0x0c : IP Interrupt Status Register (Read/TOW)
//   bit 0 - Channel 0 (ap_done)
//   others - reserved
```

Slide 4-24:

Block-Level I/O for C/RTL Cosimulation

- To use the C/RTL cosimulation feature to verify the RTL design, one or more of the following must be true
 - Top-level function must be synthesized using an `ap_ctrl_hs` or `ap_ctrl_chain` function-level interface
 - Design must be purely combinational
 - Top-level function must have an initiation interval of 1
 - Interface must be all arrays that are streaming and implemented with `ap_fifo`, `ap_hs`, or `axis` interface modes
- If one of these conditions is not met, C/RTL cosimulation halts with the following message

```
@E [SIM-345] Cosim only supports the following 'ap_ctrl_none' designs: (1) combinational designs; (2) pipelined design with task interval of 1; (3) designs with array streaming or hls_stream ports.
@E [SIM-4] *** C/RTL co-simulation finished: FAIL ***
```

Slide 4-25:

Port-Level I/O Protocols

- Overview of Vivado HLS Tool I/O Ports and Protocols
- Block-Level I/O Protocols
- **Port-Level I/O Protocols**
- DATA_PACK Directive
- Summary



Slide 4-26:

Port-Level I/O Protocols



- You have seen how the function return is validated
 - Block-level output signal ap_done goes high to indicate the function return is valid
 - This allows downstream blocks to correctly sample the output port



- For other outputs, port-level I/O protocols can be added
 - Allowing upstream and downstream blocks to synchronize with the other data ports
 - Type of protocol depends on the type of C port
 - Pass-by-value scalar
 - Pass-by-reference pointers (& references in C++)
 - Pass-by-reference arrays

Starting point is the type of argument used by the C function

Slide 4-27:

Let's Look at an Example



```
#include "adders.h"
int adders(int in1, int in2, int *sum) {
    int temp;
    *sum = in1 + in2 +
    *sum;
    temp = in1 + in2;
    return temp;
}
```

"Sum" is a pointer which is read and written to : an Inout

Argument Type	Scalar		Array			Pointer or Reference		
	pass-by-value		pass-by-reference			pass-by-reference		
Interface Mode	Input	Return	I	IO	O	I	IO	O
ap_ctrl_none								
ap_ctrl_hs		D						
ap_ctrl_chain								
axis								
s_axilite								
m_axi								
ap_none	D					D		
ap_stable								
ap_ack								
ap_vld								
ap_ovld						D		
ap_hs							D	
ap_memory			D	D	D			
bram								
ap_fifo								
ap_bus								

The port for "Sum" can be any of these interface types

The default for port for "Sum" will be type ap_ovld

Now, let's look at what these interfaces are ...

Key:
I : input
IO : inout
O : output
D : Default Interface

Supported. D = Default Interface

Not Supported



From 2014.3 onwards, native AXI4 Lite Slave interfaces support memory interfaces.

Slide 4-28:

Interface Types



- Multiple interface protocols are available
 - Every combination of C argument and port protocol is **not** supported
 - Code modification may be required to implement a specific I/O protocol

Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
<code>ap_ctrl_none</code>									
<code>ap_ctrl_hs</code>		D							
<code>ap_ctrl_chain</code>									
<code>axis</code>									
<code>s_axilite</code>									
<code>m_axi</code>									
<code>ap_none</code>	D					D			
<code>ap_stable</code>									
<code>ap_ack</code>									
<code>ap_vld</code>							D		
<code>ap_ovld</code>							D		
<code>ap_hs</code>									
<code>ap_memory</code>			D	D	D				
<code>bram</code>									
<code>ap_fifo</code>				D					D
<code>ap_bus</code>									

 Supported
 D = Default Interface
 Not Supported



Block-level protocols can be applied to the return port, but the port can be omitted and just the function name specified.

Slide 4-29:

Default I/O Protocols



- Default port protocols
 - Inputs: ap_none
 - Outputs: ap_vld
 - Inout: ap_ovld
 - In port gets ap_none
 - Out port gets ap_vld
 - Arrays: ap_memory
 - All shown as the default (D) on previous slide
- Result of the default protocols
 - No protocol for input ports
 - They should be held stable for the entire transaction
 - There is no way to know when they will be read
 - Output writes have an accompanying output valid signal that can be used to validate them
 - Arrays will default to RAM interfaces (two-port RAM is the default RAM)

```
@I [RTGEN-500] Setting IO mode on port 'adders|in1' to 'ap_none'.
@I [RTGEN-500] Setting IO mode on port 'adders|in2' to 'ap_none'.
@I [RTGEN-500] Setting IO mode on port 'adders|in3' to 'ap_none'.
@I [RTGEN-500] Setting IO mode on port 'adders|return' to 'ap_ctrl_hs'.
```

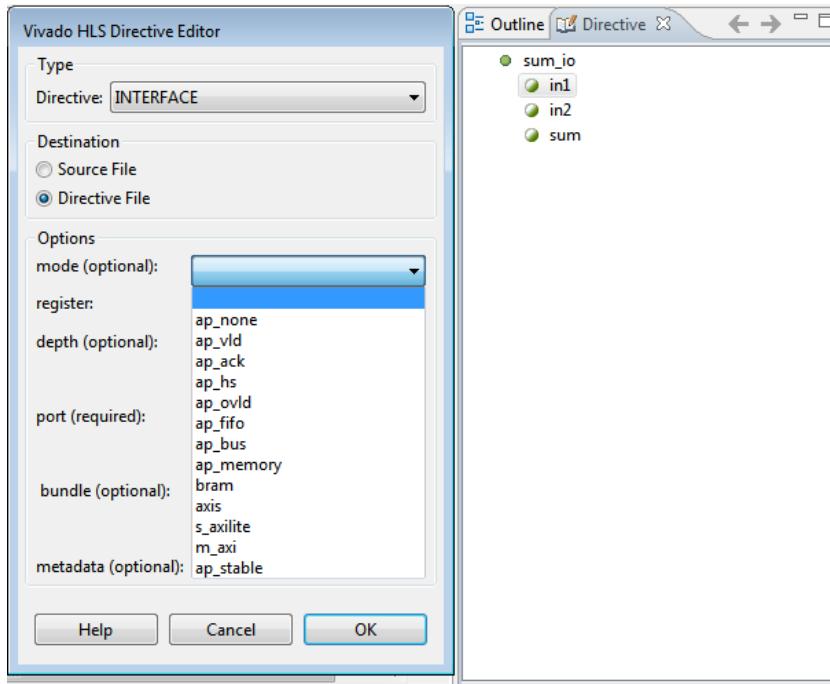
Vivado HLS tool shell/console always shows the results of interface synthesis

Slide 4-30:

Specifying I/O Protocols



- Select the port in the Directives pane to specify a protocol
 - Select the port
 - Right-click and select **Interface**
 - Select the protocol from the drop-down list
- Or apply using Tcl or Pragma



Slide 4-31:

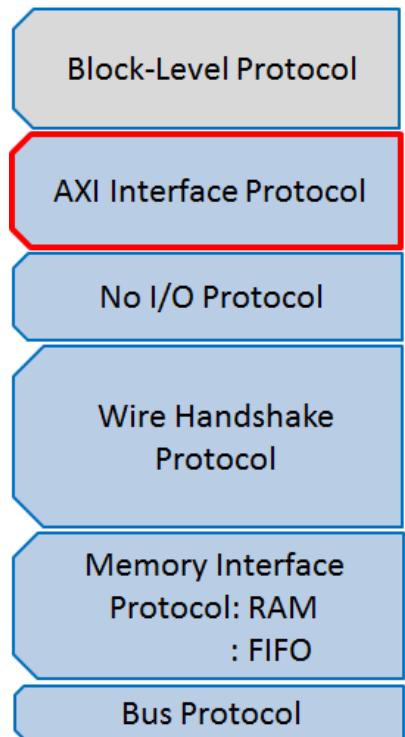
Port-Level I/O Protocols: AXI Interfaces



- Overview of Vivado HLS Tool I/O Ports and Protocols
- Block-Level I/O Protocols
- **Port-Level I/O Protocols**
 - **Port-Level I/O Protocols: AXI Interfaces**
 - Port-Level I/O Protocols: No I/O Protocol
 - Port-Level I/O Protocols: Memory Interfaces
- DATA_PACK Directive
- Summary

Slide 4-32:

Interface Type: AXI Interfaces



Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld							D		
ap_ovld							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo								D	
ap_bus									



Supported D = Default Interface



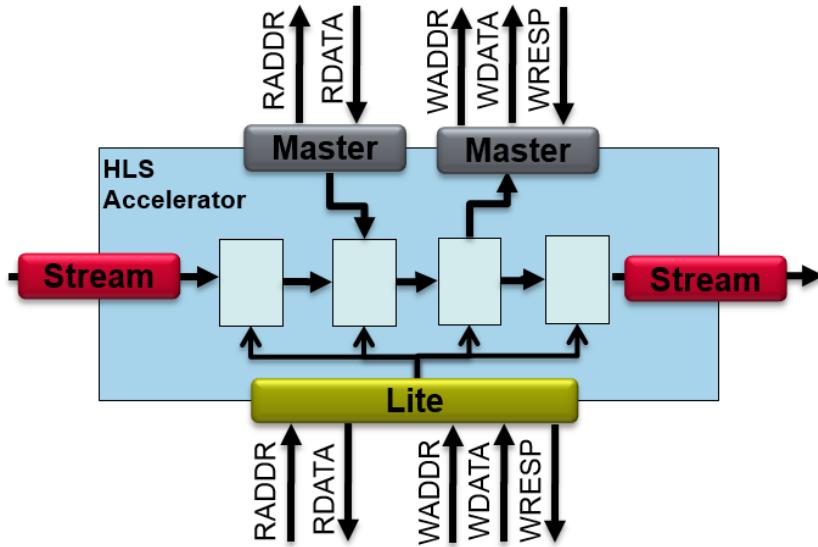
Not Supported

Slide 4-33:

Using the AXI Interface



- All three AXI interfaces are supported by the INTERFACE directive
 - AXI4-Master
 - AXI4-Lite (Slave)
 - AXI4-Stream
- Provided in RTL after synthesis
- Supported by the C/RTL cosimulation



Slide 4-34:

Interfacing to the PS in the Zynq All Programmable SoC



- ARM® processor ports in combination with HLS accelerator AXI4 ports

	AXI4-Lite	AXI-Stream (plus AXI-DMA)	AXI4-Master
GP Ports	Configuration and control of the IP from the ARM processor	N/A	N/A
ACP Ports	N/A	High-speed data transfer via AXI-DMA*	High-speed data transfer with low latency*
HP Ports	N/A	High-speed data transfer via AXI-DMA	High-speed bursts optimized for throughput

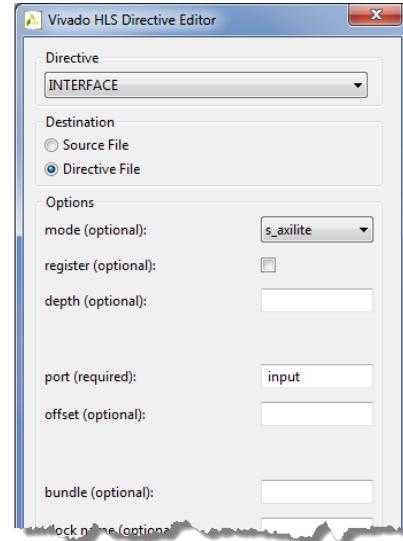
* Ideal if data fits in cache and if PS produces or consumes PL data

Slide 4-35:

AXI4-Slave Lite Interface



- Interface mode: `s_axilite`
 - Supported with the `INTERFACE` directive
 - Multiple ports can be grouped into the same AXI4-Lite slave interface
 - All ports which use the same bundle name are grouped
 - Reset is automatically implemented as active Low
- Grouped ports
 - Default mode is `ap_none` for input ports
 - Default mode is `ap_vld` for output ports
 - Default mode `ap_ctrl_hs` for function (return port)
 - Default mode can be changed with additional `INTERFACE` directives



```
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;
}
```



The AXI4-Lite slave clock and reset signals are called `ap_clk` and `ap_rst_n` in the exported IP. When the AXI4-Lite slave interface is selected, the reset is automatically implemented as active Low. This is the requirement to conform to the AXI4 standard. If an active High reset was selected or the default used, it is overridden.

The clock option adds a separate clock for AXI-Lite interfaces. The default, without the clock option, is to use the same clock as the design.

Keep the following in mind when you are considering using the AXI-Lite clock:

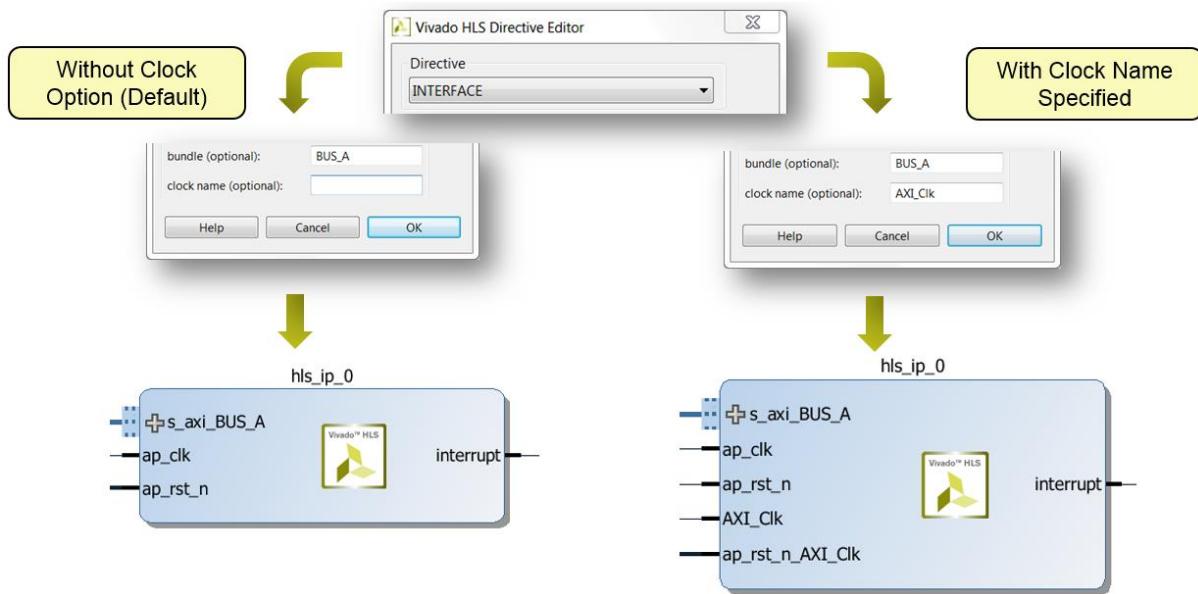
- It is synchronous with the primary clock; there are no FIFOs between clock regions.
- It must be same frequency or slower than the primary clock.
- No checking is performed inside HLS; you must wire correctly in the IPI or RTL design.

Slide 4-36:

AXI4-Lite Interface: Separate Clock and Reset



- Separate clock and reset for AXI-Lite interfaces
 - New clock option adds a separate clock for AXI-Lite interfaces
 - The default, without a clock option, is to use the same clock as the design



Slide 4-37:

AXI4-Lite Interface: Clock Details



- AXI-Lite clock assumptions
 - Must be the same frequency or slower than the primary clock
 - Synchronous with the primary clock
 - There are **NO** FIFOs between clock regions
 - No checking is performed inside HLS
 - Wire correctly in IPI or in the RTL design: AXI-Lite reset is always active Low
- AXI-Lite bundles
 - Clock option only required on one bundle signal (named bundle or default bundle)
 - All signals in the same bundle inherit the same AXI-Lite clock

```
// Example of 2 AXI-Lite interfaces (bundles)
```

```
// Default AXI-Lite bundle implemented with a separate clock (called AXI_clk1)
```

```
#pragma HLS interface s_axilite port=a clock=AXI_clk1
#pragma HLS interface s_axilite port=b
```

```
// CTRL1 AXI-Lite bundle implemented with a separate clock (called AXI_clk2)
```

```
#pragma HLS interface s_axilite port=c bundle=CTRL1 clock=AXI_clk2
#pragma HLS interface s_axilite port=d bundle=CTRL1
```

Slide 4-38:

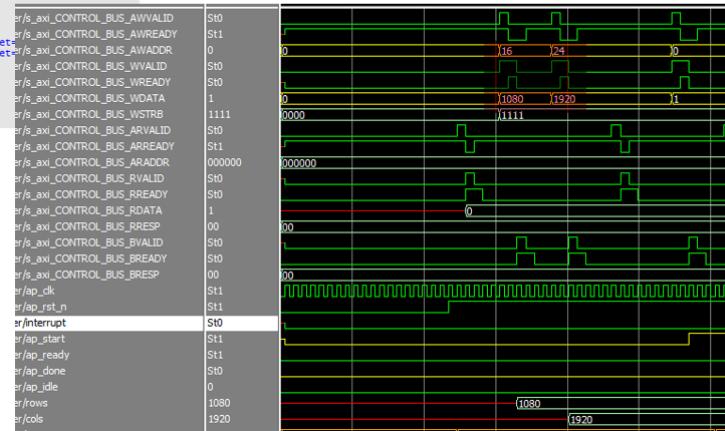
AXI4-Lite Interface: Cosimulation



- RTL cosimulation supports the AXI-Lite interface
- AXI-Lite interface results in write access during RTL cosimulation

```

19 // AXI stream version
20 AXI_STREAM src_axi, dst_axi;
21
22 // Convert OpenCV format to AXI4 Stream format
23 IPImage2AXIVideo(src, src_axi);
24
25 // DUT: the function to be synthesized
26 //for (int i=0;i<1;i++)
27 //    image_filter(src_axi, dst_axi, src->height, src->width);
28
29 // Convert the AXI4 Stream data to OpenCV format
30 AXIVideo2IPImage(dst_axi, dst);
31
32 // Standard OpenCV image functions
33 cvSaveImage(OUTPUT_IMAGE, dst);
34
35 cvReleaseImage(&src);
36 vReleaseImage(&dst);
image_filter.cpp
  
```

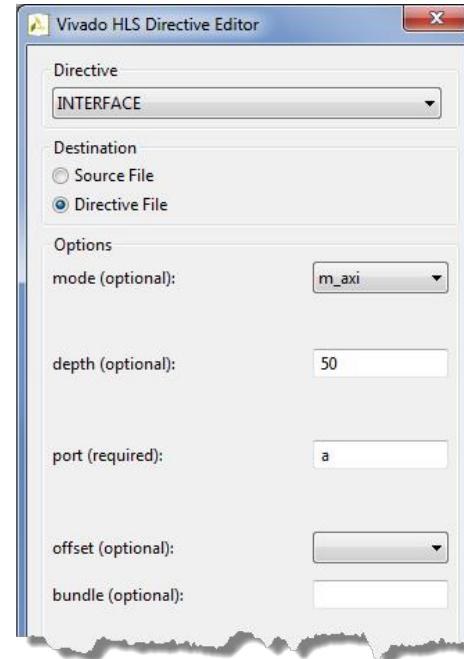


Slide 4-39:

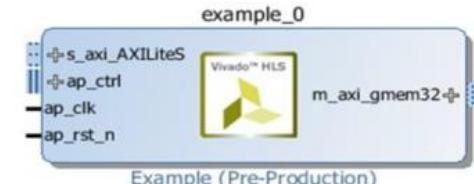
AXI4-Master



- Interface Mode: `m_axi`
 - Supported with the INTERFACE directive
- Options
 - Multiple ports can be grouped into the same AXI4-Master interface
 - All ports which use the same bundle name are grouped
 - Depth option is required for the C/RTL cosimulation
 - Required for pointers, not arrays
 - Set to the number of values read/written
 - Option to support offset or base address



```
void example(volatile int *a)
{
    #pragma HLS INTERFACE m_axi depth=50 port=a
}
```



Example (Pre-Production)

Slide 4-40:

AXI4-Master: Example

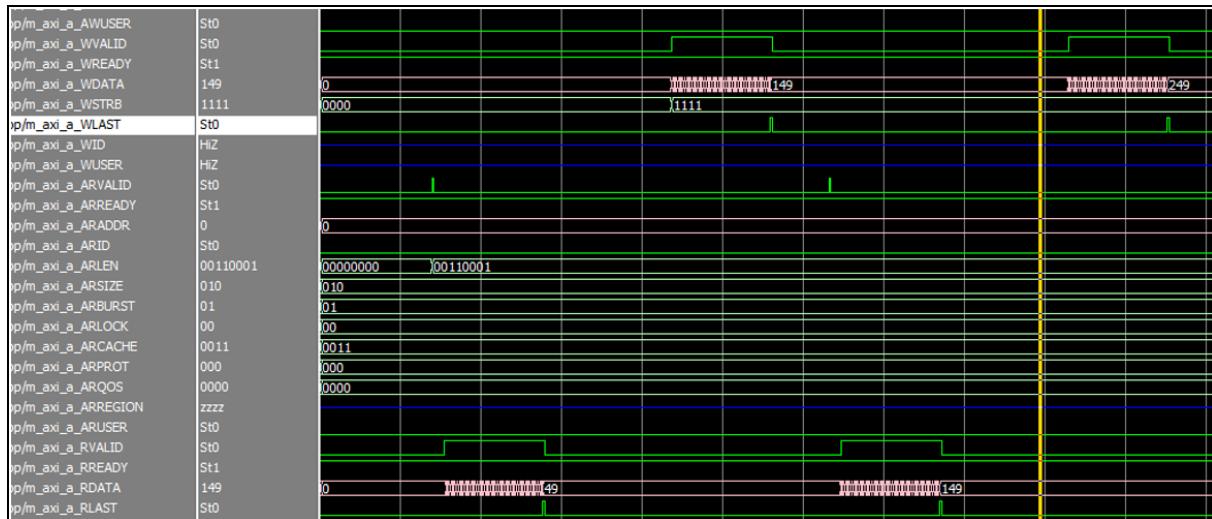


- memcpy example
 - memcpy is used to burst-access read/write data
 - Block-level I/O (all the ap* inputs/outputs) are memory mapped in the AXI4-Lite interface

```

3 void memcpy_top(volatile int *a) {
4 // Port a is assigned to an AXI4 master interface
5 #pragma HLS INTERFACE m_axi depth=50 port=a
6 #pragma HLS INTERFACE s_axilite port=return bundle=AXILiteS
7
8 int i;
9 int buff[50];
10
11 // memcpy creates a burst access to memory
12 memcpy(buff,(const int *)a,50*sizeof(int));
13
14 for (i=0; i < 50; i++){
15 #pragma HLS pipeline rewind
16     buff[i] = buff[i] + 100;
17 }
18
19 memcpy((int *)a,buff,50*sizeof(int));
20
21 }
22

```



Slide 4-41:

AXI4-Master: Burst Access



- Burst accesses inferred from *for* loops
 - Performed automatically by the Vivado HLS tool
- Requirements
 - Loop must be pipelined
 - Addresses must be accessed in increasing order
 - Memory accesses cannot be guarded by a conditional statement
 - Do not flatten nested loops
 - Only one read and write per AXI port allowed in a *for* loop

```
//Port a is assigned to an AXI4 master interface
void example(volatile int *a){
#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE s_axilite port=return

int i;
int buff[50];

// Burst data into the Design
//memcpy creates a burst access to memory
memcpy(buff,(const int*)a,50*sizeof(int));

// Perform some Calculation
for(i=0; i < 50; i++){
    buff[i] = buff[i] + 100;
}

// Burst Data out of the Design
//Alternatively, for loop creates a burst access to memory
for(i=0; i < 50; i++)
#pragma HLS PIPELINE
    a[i] = buff[i];
}
```

Slide 4-42:

AXI4-Master: Burst Access with Multiple Ports



- Multiple access bursts
 - Only one access (read or write) per port can be inferred from a *for* loop
 - No simultaneous read and write access
 - If multiple ports are used, multiple read or writes can be performed

```
// Two pointers are accessed
void example(volatile int *a, int *b){
#pragma HLS INTERFACE s_axilite port=return

// Two different AXI ports are used
#pragma HLS INTERFACE m_axi depth=50 port=a
#pragma HLS INTERFACE m_axi depth=50 port=b bundle=d2_port
int i;
int buff[50];

//Copy data in
for(i=0; i < 50; i++)
#pragma HLS PIPELINE
    // Separate AXI ports mean both a and b reads implemented as bursts
    buff[i] = a[i] + b[i];
}
...
}
```

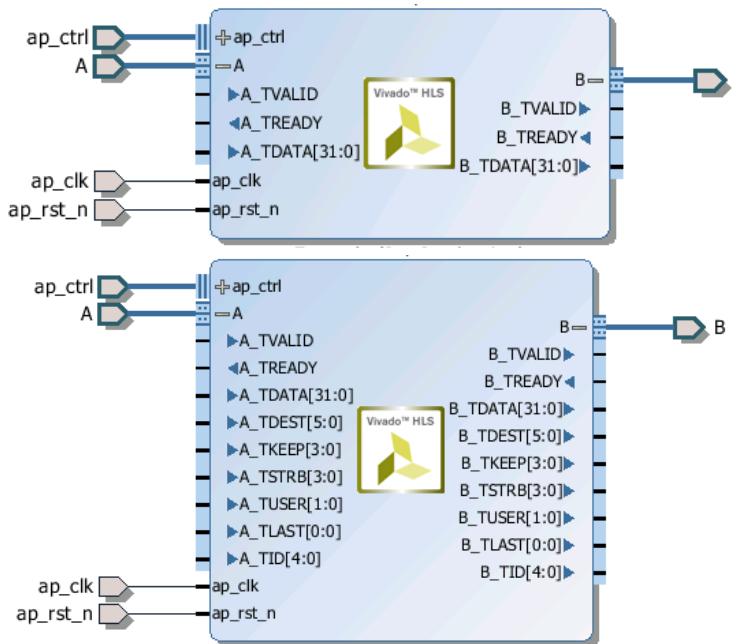
Slide 4-43:

AXI4-Stream Interface



- AXI Stream interface
 - Can be applied on any input argument
 - Can be applied on array or pointer output argument

- Two ways to implement AXI Stream interface
 - AXI4-Stream without side channels
 - Supported with INTERFACE directive **axis**
 - AXI4-Stream with side channels
 - Side channels are optional signals but part of the AXI4-Stream standard
 - ap_axi_sdata.h header file contains structs to support it



An AXI4-Stream interface can be applied to any input argument and any array or pointer output argument. Since an AXI4-Stream interface transfers data in a sequential streaming manner, it cannot be used with arguments which are both read and written. An AXI4-Stream interface is always sign-extended to the next byte. For example, a 12-bit data value is sign-extended to 16-bit.

AXI4-Stream interfaces can be implemented as registered interfaces or non-registered interfaces. This is controlled using the INTERFACE directive `-register` option. The default is a non-registered interface. A non-registered AXI4-Stream interface implies a combinational path from TVALID to TREADY for inputs, and from TREADY to TVALID for outputs.

Be careful of connecting a non-registered AXI4-Stream output port to a non-registered AXI4-Stream input port, as this creates a combinational loop in the final design. The design will still be implemented, but DRC warnings will also be generated.

Slide 4-44:

AXI4-Stream: Interface Registers



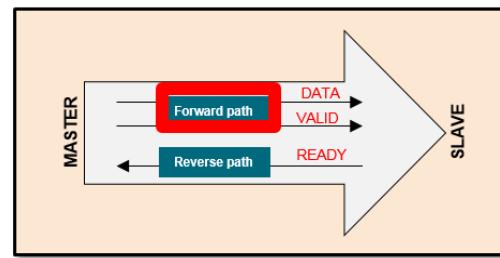
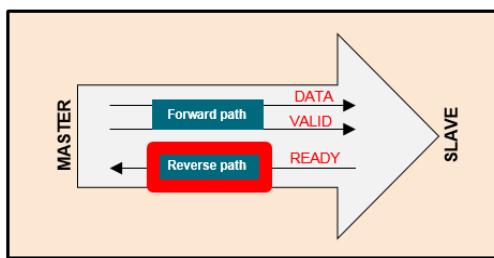
- AXI4-Stream registered
 - Change to register option on AXI4-Stream interfaces
 - Eliminates timing loops during design integration

```
#pragma HLS interface axis port=a register
```

- Register default behavior
 - Default: No registers

Inputs: The reverse path (ready) is registered

Outputs: The forward path (data & valid) is registered



Slide 4-45:

Port-Level I/O Protocols: AXI Interfaces

- Overview of Vivado HLS Tool I/O Ports and Protocols
- Block-Level I/O Protocols
- **Port-Level I/O Protocols**
 - Port-Level I/O Protocols: AXI Interfaces
 - **Port-Level I/O Protocols: No I/O Protocol**
 - Port-Level I/O Protocols: Memory Interfaces
- DATA_PACK Directive
- Summary



Slide 4-46:

Interface Type: No I/O Protocol



Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld							D		
ap_ovid							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo									D
ap_bus									

Supported D = Default Interface
 Not Supported

Slide 4-47:

No I/O Protocol



- AP_NONE: Default protocol for input ports
 - Protocol ap_none means that no additional protocol signals are added
 - Port will be implemented as just a data port
 - Except arrays that must be a RAM or FIFO interface

- AP_STABLE: ap_none with fanout benefits
 - ap_stable protocol is used to specify that a port remains constant between successive reset operations
 - Port is not set to a constant: this could be optimized away
 - Vivado HLS tool however knows there is no need to register the port fanout
 - Useful for **configuration ports**
 - Vivado HLS tool will assume the port is not updated between reset operations

Slide 4-48:

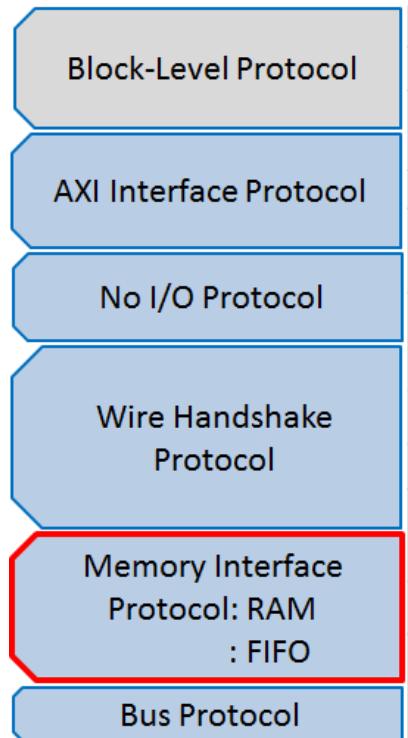
Port-Level I/O Protocols: Memory Interfaces

- Overview of Vivado HLS Tool I/O Ports and Protocols
- Block-Level I/O Protocols
- **Port-Level I/O Protocols**
 - Port-Level I/O Protocols: No I/O Protocol
 - Port-Level I/O Protocols: AXI Interfaces
 - **Port-Level I/O Protocols: Memory Interfaces**
- DATA_PACK Directive
- Summary



Slide 4-49:

Interface Type: Memory Interface



Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld							D		
ap_ovld							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo								D	
ap_bus									



Supported D = Default Interface



Not Supported

Slide 4-50:

Memory I/O Protocols



- Memory protocols can be inferred from array and pointer arguments
 - Array arguments can be synthesized to RAM or FIFO ports
 - When FIFOs specified on array ports, the ports must be read only or write only
 - Pointer (and references in C++) can be synthesized to FIFO ports
- RAM ports
 - Support arbitrary/random accesses
 - Can be implemented with dual-Port RAMs to increase the bandwidth
 - Default is a dual-port RAM interface
 - Requires two cycles read access: generate address, read data
 - Pipelining can reduce this overhead by overlapping generation with reading
- FIFO ports
 - Require read and writes to be sequential/streaming
 - Always uses a standard FIFO (single-port) model
 - Single cycle for both reads and writes

Slide 4-51:

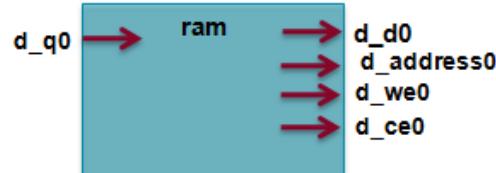
Memory I/O Protocols: Ports Generated



- RAM ports (`ap_memory`)
 - Created by protocol `ap_memory`
 - Given an array specified on the interface
 - Ports are generated for data, address, and control
 - Example shows a single port RAM
 - Dual-port resource will result in dual-port interface
 - Specify the off-chip RAM as a resource
 - Use the RESOURCE directive on the array port

- FIFO ports (`ap_fifo`)
 - Created by protocol `ap_fifo`
 - Can be used on arrays, pointers, and references
 - Standard read/write, full/empty ports generated
 - Must use separate arrays for read and write
 - Pointers/references: split into in and out ports

```
#include "ram.h"
void ram (int d[DEPTH], ...){
  ...
}
```



```
#include "fifo.h"
void fifo (int d_o[DEPTH],
           int d_i[DEPTH]) {
  ...
}
```

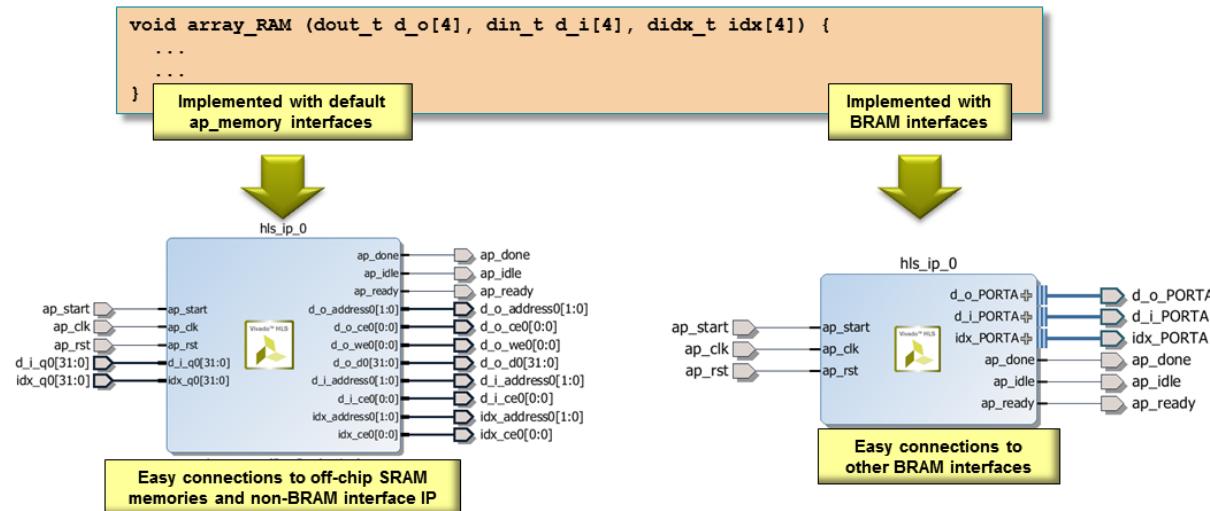


Slide 4-52:

Block RAM Interface: Ports Generated

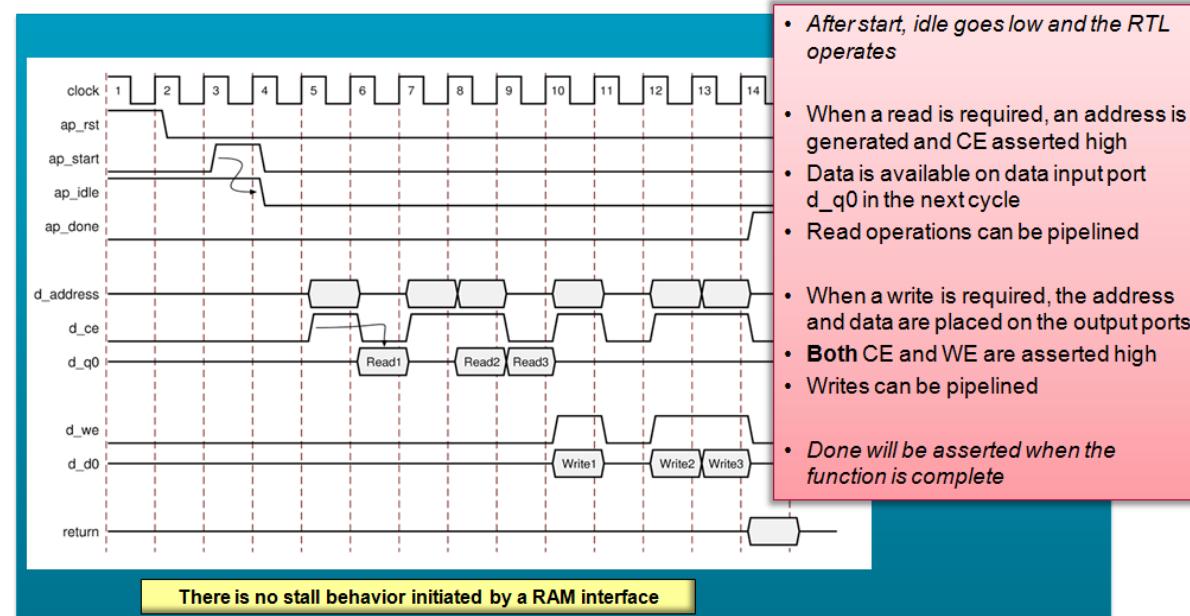


- Block RAM interface mode (BRAM)
 - Provides the same block-RAM interface as `ap_memory`
 - Grouped as a single port interface in IP integrator



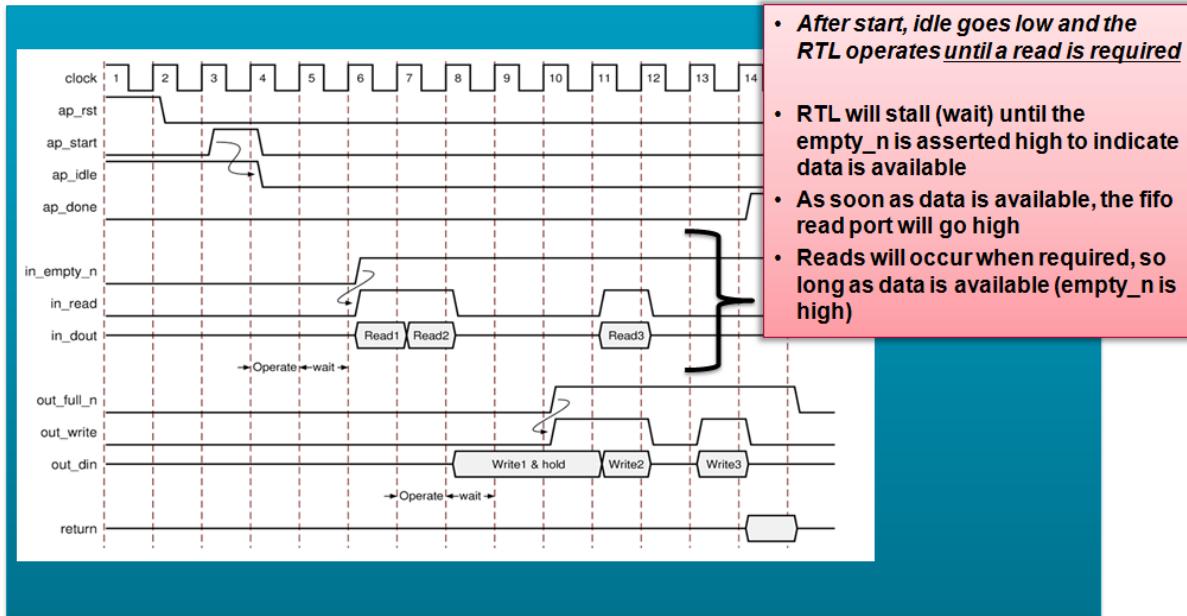
Slide 4-53:

Memory I/O Protocol (ap_memory)



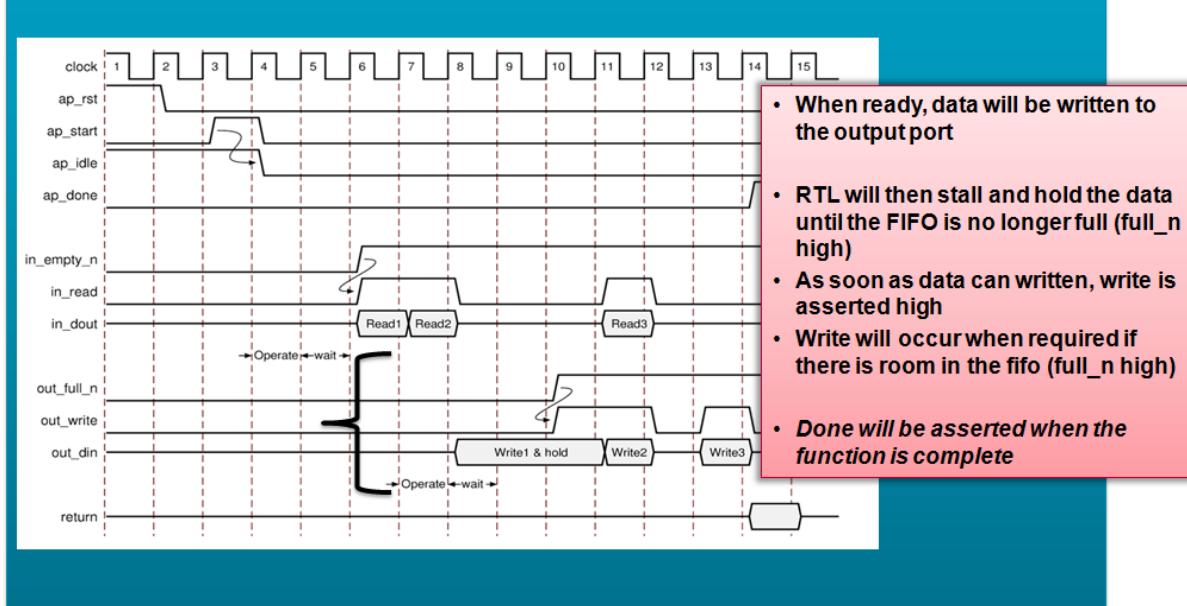
Slide 4-54:

FIFO I/O Protocol (ap_fifo, Read)



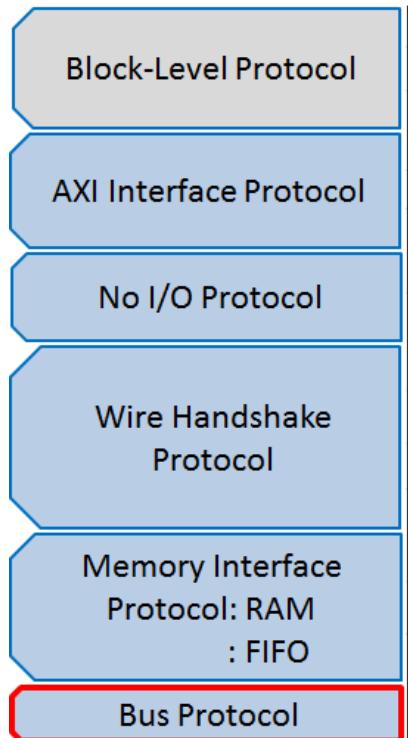
Slide 4-55:

FIFO I/O Protocol (ap_fifo, Write)



Slide 4-56:

Interface Type: Bus Protocol



Argument Type	Scalar		Array			Pointer or Reference			HLS:: Stream
Interface Mode	Input	Return	I	I/O	O	I	I/O	O	I and O
ap_ctrl_none									
ap_ctrl_hs		D							
ap_ctrl_chain									
axis									
s_axilite									
m_axi									
ap_none	D					D			
ap_stable									
ap_ack									
ap_vld							D		
ap_ovid							D		
ap_hs									
ap_memory			D	D	D				
bram									
ap_fifo								D	
ap_bus									



Supported D = Default Interface



Not Supported

Slide 4-57:

Bus I/O Protocol



- Vivado HLS tool supports a bus I/O protocol
 - I/O protocol is that of a generic bus
 - I/O protocol is not an industry standard
 - Vivado HLS tool bus protocol allows connecting to adapter cores
- Bus /IO protocol supports `memcpy`
 - Bus I/O protocol supports the C function `memcpy`
 - Provides a high-performance interface for bursting data in a DMA-like fashion
- Bus I/O protocol supports complex pointer arithmetic at the I/O
 - Pointers at the I/O can be synthesized to `ap_fifo` or `ap_bus`
 - If using `ap_fifo`, the accesses must be sequential
 - If pointer arithmetic is used, the port must use `ap_bus`

Slide 4-58:

Bus I/O and Pointer Arithmetic

- Given the following code



```
#include "bus.h"
void foo(int*d) {
    static int acc = 0;
    int i;
    for(i=0;i<4;i++){
        acc += *(d+i+1);
        *(d+i) = acc;
    }
}
```

Plus
Testbench

```
# int main () {
    int d[5];
    int i;
    for(i=0;i<5;i++){
        d[i] = i+5;
    }
    foo(d);
    // Check results
    ...
}
```

Results

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

Output address 4 is
never written to

- Only a bus interface can support the pointer arithmetic
 - All FIFO reads (writes) must be sequential and start at 0 (not i+1)
- If a FIFO interface is used
 - RTL cosimulation result

Expected	Actual
-----	-----
res[0]: 6	!= d[0]: 5
res[1]:13	!= d[1]: 6
res[2]:21	!= d[2]: 7
res[3]:30	!= d[3]: 8
res[4]: 9	== d[4]: 9

SystemC: simulation stopped by user.
 @E [SIM-3] Simulation failed: testbench return error code "1".
 @E [SIM-1] *** AutoSim finished: FAIL ***

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

Slide 4-59:

Standard and Burst Mode



- Standard mode
 - Each access to the bus results in a request then a read or write operation
 - Multiple read or writes can be performed in a single transaction

Single read and write in standard mode

```
void foo (int *d) {  
    static int acc = 0;  
    int i;  
  
    acc += d[i];  
    d[i] = acc;  
}
```

Multiple reads and writes in standard mode

```
#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;  
    }  
}
```

```
#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;  
    }  
}
```

- Burst mode

- Use the C `memcpy` function
- Copies data between array and a pointer argument
- Pointer argument can be a bus interface
 - This example uses a size of 4
 - This is more efficient for higher values

Burst Mode

```
void foo (int *d) {  
    int buf1[4], buf2[4];  
    int i;  
  
    memcpy(buf1,d,4*sizeof(int));  
  
    for (i=0;i<4;i++) {  
        buf2[i] = buf1[3-i];  
    }  
  
    memcpy(d,buf2,4*sizeof(int));  
}
```

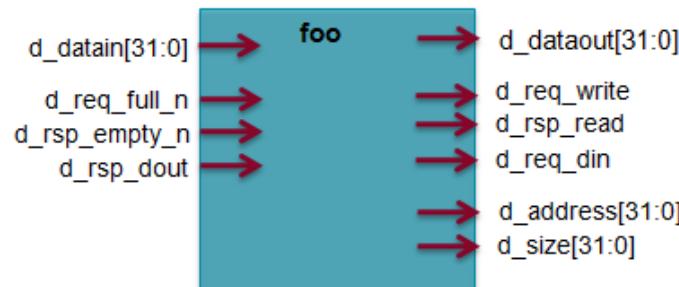
Slide 4-60:

Bus I/O Protocol: Port Generated



- Bus request then access protocol
 - Protocol will request a read/write bus access
 - Then read or write data in single or burst mode

```
#include "foo.h"
void foo (int *d) {
  ...
}
```



InputPorts	Description	OutputPorts	Description
d_datain	Input data.	d_dataout	Output data.
d_req_full_n	Active low signal indicates the bus bridge is full. The design will stall, waiting to write.	d_req_write	Asserted to initiate a bus access.
d_rsp_empty_n	Active low signal indicates the bus bridge is empty and can accept data.	d_req_din	Asserted if the bus access is to write. Remains low if the access is to read.
Standard mode: Address port gives the index address – e.g., "(d+i), addr = value of "i"			d_rsp_read
			Asserted to start a bus read (completes a bus access request and starts reading data)
		d_address	Offset for the base address.
		d_size	Indicates the burst (read or write) size.

Slide 4-61:

DATA_PACK Directive



- Overview of Vivado HLS Tool I/O Ports and Protocols
- Block-Level I/O Protocols
- Port-Level I/O Protocols
- **DATA_PACK Directive**
- Summary

Slide 4-62:

DATA_PACK (1)

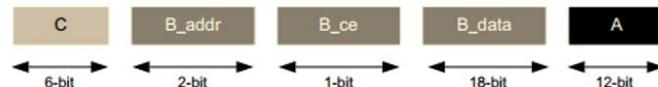


- By default, the members of struct are implemented as individual ports
- Using DATA_PACK results in a single wide port

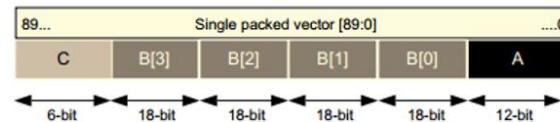
```
typedef struct{
    int12 A;
    int18 B[4];
    int6 C;
} my_data;
```

```
void foo(my_data *a)
```

Struct Port Implementation



DATA_PACK optimization



The DATA_PACK optimization directive is used for packing all the elements of a struct into a single wide vector. This allows all members of the struct to be read and written to simultaneously.

The member elements of the struct are placed into the vector in the order they appear in the C code: the first element of the struct is aligned on the LSB of the vector and the final element of the struct is aligned with the MSB of the vector. Any arrays in the struct are partitioned into individual array elements and placed in the vector from lowest to highest, in order.

Care should be taken when using the DATA_PACK optimization on structs with large arrays. If an array has 4096 elements of type int, this will result in a vector (and port) of width $4096 \times 32 = 131072$ bits. The Vivado HLS tool can create this RTL design; however, it is very unlikely that logic synthesis will be able to route this during the FPGA implementation.

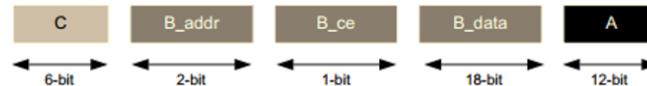
Slide 4-63:

DATA_PACK (2)

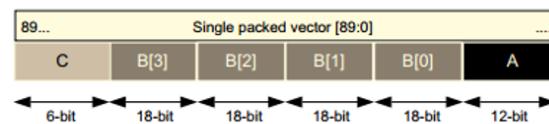


- Using DATA_PACK results in a single wide port
 - struct_level byte padding aligns entire struct to the next 8-bit boundary
 - field_level byte padding aligns each struct member to the next 8-bit boundary

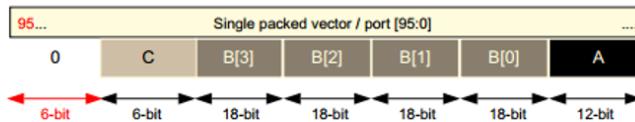
Struct Port Implementation



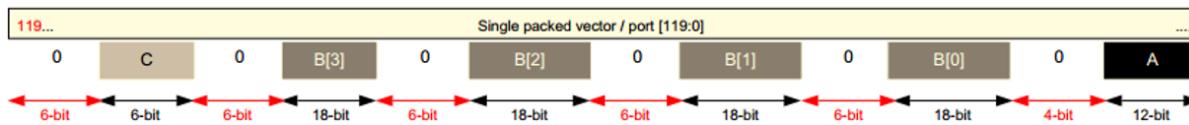
DATA_PACK optimization



DATA_PACK optimization with byte_pad on the struct_level



DATA_PACK optimization with byte_pad on the field_level



If a struct port using DATA_PACK is to be implemented with an AXI4 interface you may want to consider using the DATA_PACK byte_pad option. The byte_pad option is used to automatically align the member elements to 8-bit boundaries. This alignment is sometimes required by Xilinx IP. If an AXI4 port using DATA_PACK is to be implemented, refer to the documentation for the Xilinx IP it will connect to and determine if byte alignment is required.

Slide 4-64:

Summary

- Overview of Vivado HLS Tool I/O Ports and Protocols
 - Block-Level I/O Protocols
 - Port-Level I/O Protocols
 - DATA_PACK Directive
 - **Summary**
- 

Slide 4-65:

Summary



- Overview of Vivado HLS tool I/O ports, protocols and options
 - Three types of I/O selection: block, port, and bus interfaces
- Block-level protocols
 - Default handshakes at the block level
- Port-level protocols
 - Wide support for all standard I/O protocols
 - Protocol is dependent on the C variable type (pointer, etc.)
- Creating bus interfaces
 - Support for AXI interfaces
 - Assign as an external resource, just like a RAM
 - Choice of adapter is a function of the C variable type (pointer, etc.)

Capture Your Notes Here



Pipelining for Performance

Slide 5-1:

2016.1

This module describes various techniques for improving the throughput of a design.
45 minutes.

Slide 5-2:

Objectives

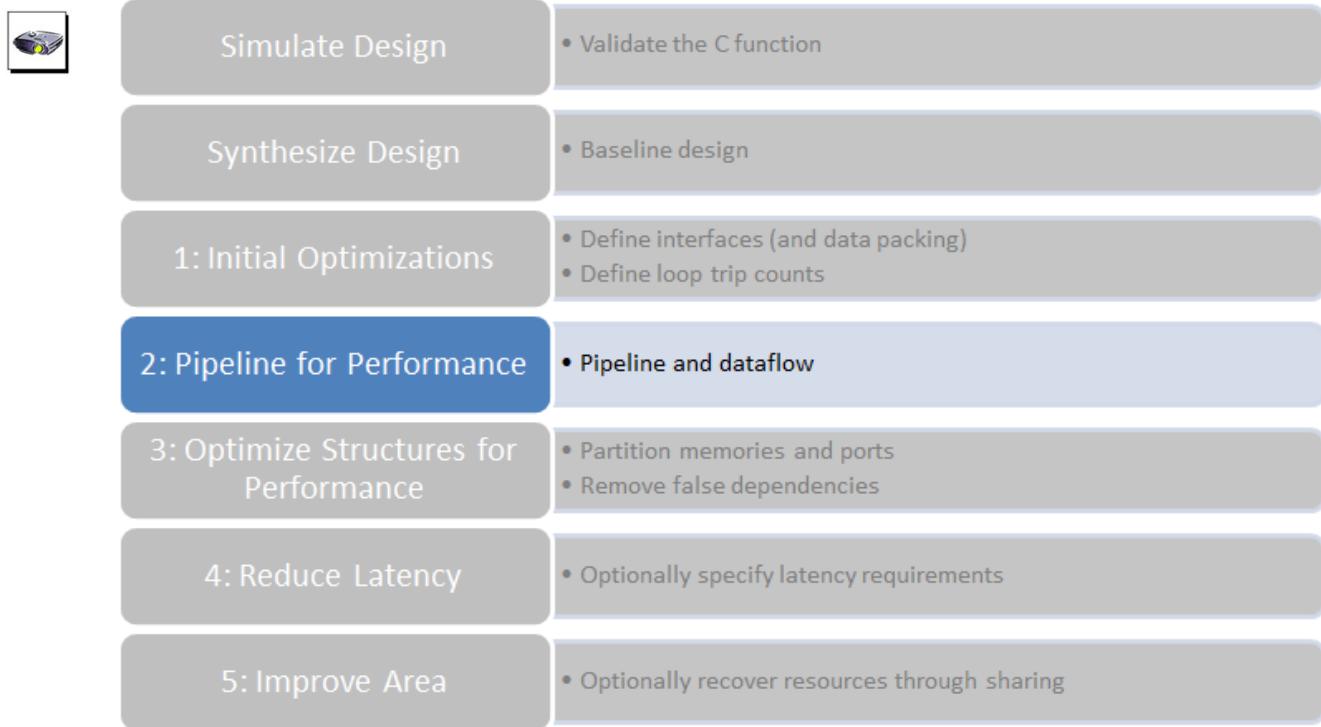


After completing this module, you will be able to:

- Describe the pipelining technique that improves throughput of a design
- Recognize the dataflow technique that improves throughput of a design
- Identify some of the bottlenecks that impact design performance
- Use coding techniques to overcome performance bottlenecks

Slide 5-3:

HLS UltraFast Design Methodology



The next stage in creating a high performance design is to pipeline the functions, loops, and tasks.

Slide 5-4:

Step 2: Pipeline for Performance



Directives and Configurations	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
DATAFLOW	Enables task-level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.
RESOURCE	Specifies a resource (core) to use to implement a variable (array, arithmetic operation, or function argument) in the RTL.
Config Compile	Allows loops to be automatically pipelined based on their iteration count.



At this stage of the optimization process you want to create as much concurrent operation as possible. You can apply the PIPELINE directive to functions and loops. You can also use the DATAFLOW directive at the level that contains the functions and loops to make them work in parallel.

A recommended strategy is to work from the bottom up and be aware of the following:

- Some functions and loops contain sub-functions. If the sub-function is not pipelined, the function above it might show limited improvement when it is pipelined. The non-pipelined sub-function will be the limiting factor.
- Some functions and loops contain sub-loops. When you use the PIPELINE directive, the directive automatically unrolls all loops in the hierarchy below. This can create a great deal of logic. It might make more sense to pipeline the loops in the hierarchy below.
- Loops with variable bounds cannot be unrolled, and any loops and functions in the hierarchy above these loops cannot be pipelined. To address this issue, pipeline these loops, and use DATAFLOW optimization to maximize the performance of the function that contains the loop. Alternatively, rewrite the loop to remove the variable bound.

Slide 5-5:

Pipelining



- **Pipelining**
- Dataflow Optimization
- Summary

Slide 5-6:

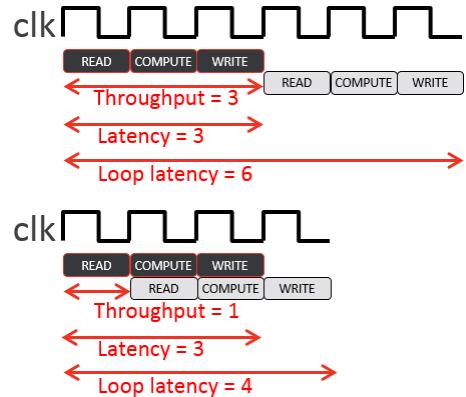
Loop Pipelining



- Pipelining allows for loop iterations to run in parallel
 - Improves throughput (=initiation interval)

```
void foo (...) {
...
add: for (i=1;i<=2;i++)
{
    op_READ;
    op_COMPUTE;
    op_WRITE;
}
...
}
```

Without Pipeline →



- Three clock cycles before operation RD can occur again
 - Throughput = three cycles
- Three cycles before the first output is written
 - Latency = three cycles
 - For the loop, six cycles
- Latency is the same
- Throughput is better
 - Fewer cycles, higher throughput
- Loop latency has been improved

Slide 5-7:

Function Pipelining

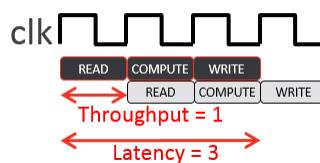


```
void foo (...) {
    op_READ;
    op_COMPUTE;
    op_WRITE;
}
```

Without Pipeline



With Pipeline



- Three clock cycles before operation RD can occur again
 - Throughput = three cycles
- Three cycles before the first output is written
 - Latency = three cycles
- Latency is the same
- Throughput is better
 - Fewer cycles, higher throughput

Slide 5-8:

Pipelining: Be Careful Where You Put It!



- Vivado® HLS tool will attempt to unroll all loops nested **below** a PIPELINE directive
 - May not succeed for various reason and/or may lead to unacceptable area
 - Pipelining the **inner-most loop** will result in best performance for area
 - Or next one (or two) out if inner most is modest and fixed
 - Outer loops will keep the inner pipeline fed

```
void foo(in1[ ][], in2[ ][], ...) {
...
L1:for(i=1;i<N;i++) {
    L2:for(j=0;j<M;j++) {
        #pragma AP PIPELINE
        out[i][j] = in1[i][j] + in2[i][j];
    }
}
}
```

1adder, 3 accesses

```
void foo(in1[ ][], in2[ ][], ...) {
...
L1:for(i=1;i<N;i++) {
    #pragma AP PIPELINE
    L2:for(j=0;j<M;j++) {
        out[i][j] = in1[i][j] + in2[i][j];
    }
}
}
```

Unrolls L2
M adders, 3M accesses

```
void foo(in1[ ][], in2[ ][], ...) {
#pragma AP PIPELINE
...
L1:for(i=1;i<N;i++) {
    L2:for(j=0;j<M;j++) {
        out[i][j] = in1[i][j] + in2[i][j];
    }
}
}
```

Unrolls L1 and L2
N*M adders, 3(N*M) accesses



Fundamentally, there are two types of C functions: those that are frame based and those that are sampled based. No matter which style is used, almost identical RTL IP can be produced in both cases: the difference is in how the optimization directives are applied. The selection of which style to use is down to you; use whatever style is the easiest for you to capture your description.

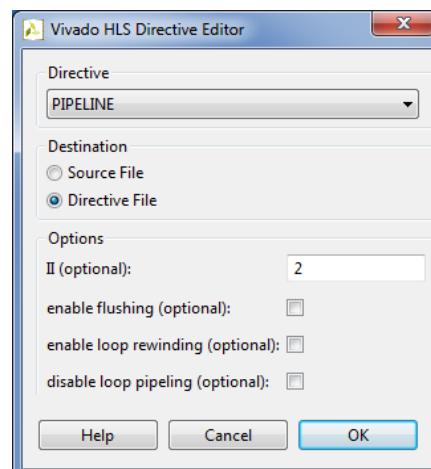
In general, pipelining the top level will not work well for frame based.

Slide 5-9:

Pipelining Commands



- Pipeline directive pipelines functions or loops
 - This example pipelines the function with an initiation interval (II) of 2
 - II is the same as the throughput but this term is used exclusively with pipelines



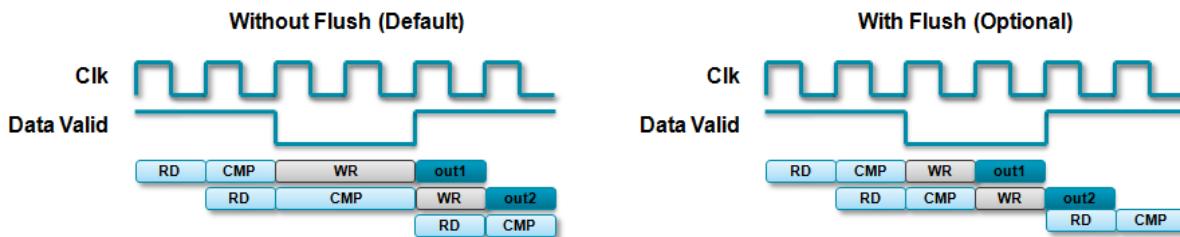
- Omit the target II and Vivado HLS tool will automatically pipeline for the fastest possible design
 - Specifying a more accurate maximum may allow more sharing (smaller area)

Slide 5-10:

Pipeline Flush



- Pipelines can optionally be flushed
 - Flush: when the input enable goes low (no more data) all existing results are flushed out
 - Input enable can be from an input interface or from another block in the design
 - Default is to stall all existing values in the pipeline



- With flush
 - When no new input reads are performed
 - Values already in the pipeline are flushed out

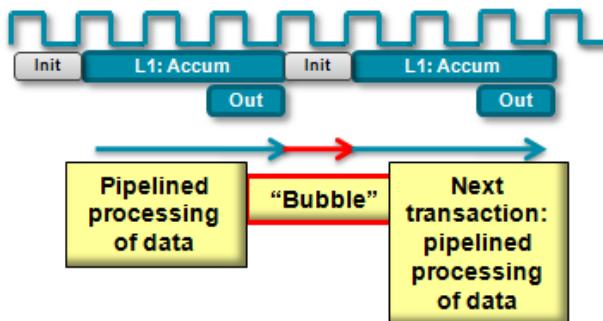
Slide 5-11:

Pipelining the Top-Level Loop



- Loop pipelining top-level loop may give a "bubble"
 - A bubble here is an interruption to the data stream
 - Given the following

```
void foo_top (in1, in2, *out1_data...){  
    accum=0;  
    ...  
    L1:for(i=1;i<N;i++){  
        accum = accum + in1 + in2;  
    }  
    *out1_data = accum;  
}
```



- Function will process a stream of data
- Next time the function is called, it still requires a state/cycle to execute the initial operations
 - These operations are any that occur before the loop starts
 - These operations may include interface start/stop/done signals
- This can result in an unexpected interruption of the data stream: a "bubble"

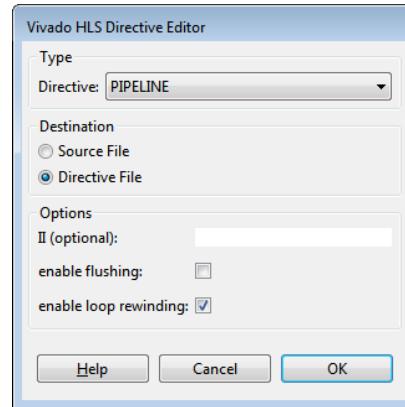
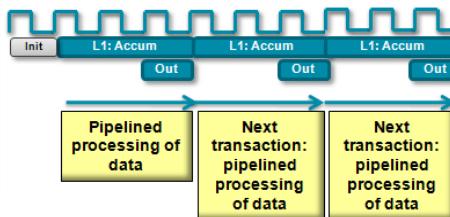
Slide 5-12:

Continuous Pipelining of the Top-Level Loop



- Use the "rewind" option for continuous pipelining
 - Immediate re-execution of the top-level loop
 - Operation rewinds the function execution to the start of the loop
 - Pushes any initialization statements before the start of the loop into the loop
 - These statements cannot contain *if-else* branches

```
void foo_top (in1, in2, *out1_data...) {
    accum=0;
    ...
    L1:for(i=1;i<N;i++) {
        accum = accum + in1 + in2;
    }
    *out1_data = accum;
}
```



- Rewind option *only* affects top-level loops

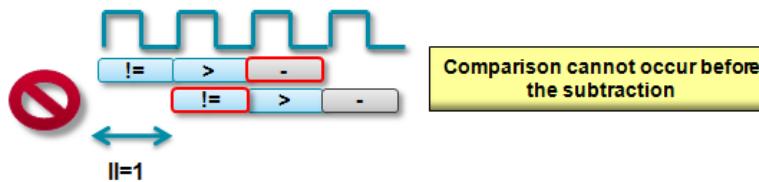
Slide 5-13:

Issues That Prevent Pipelining



- Pipelining functions unrolls all loops
 - Loops with variable bounds cannot be unrolled
 - This will prevent pipelining
 - Re-code to remove the variables bounds: max bounds with an exit
- Feedback prevent/limits pipelines
 - Feedback within the code will prevent or limit pipelining
 - Pipeline may be limited to higher initiation interval (more cycles, lower throughput)

```
void foo_top (a,b,c,d) {
    ...
    while (a != b) {
        if (a > b) {
            a = b;
        } else {
            b = a;
        }
        ...
    }
}
```



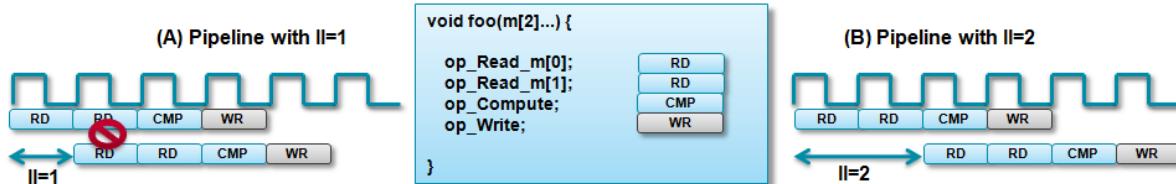
- Resource contention may prevent pipelining
 - Can occur within input and output ports/arguments
 - This is a classic way in which arrays limit performance

Slide 5-14:

Resource Contention: Unfeasible Initiation Intervals



- Sometimes the II specification cannot be met
 - In this example there are two read operations on the same port



- An II=1 cannot be implemented
 - Same port cannot be read at the same time
 - Similar effect with other resource limitations
 - For example, if functions, multipliers, etc. are limited
- Vivado HLS tool will automatically increase the II
 - Vivado HLS tool will always try to create a design, even if constraints must be violated

Slide 5-15:

Dataflow Optimization



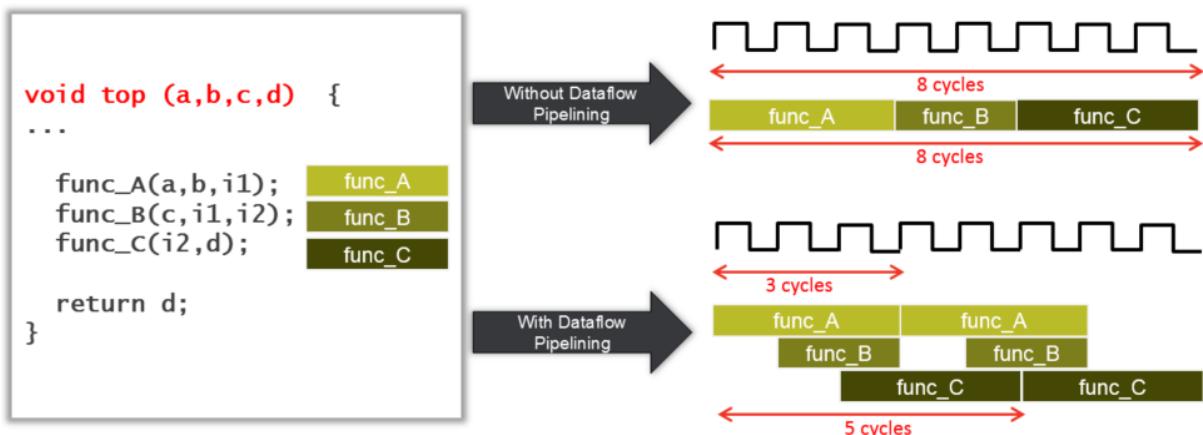
- Pipelining
- **Dataflow Optimization**
- Summary

Slide 5-16:

Dataflow



- Dataflow works like pipelining, but works on the function/loop level
- Dataflow works on packet/array of data while pipelining works on sample based



In the example **without dataflow pipelining**, the implementation requires eight cycles before a new input can be processed by func_A and eight cycles before an output is written by func_C.

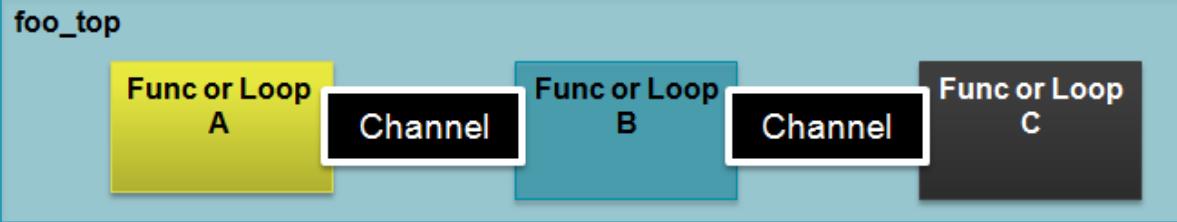
In the example **with dataflow pipelining**, func_A can begin processing a new input every three clock cycles (lower initiation interval) and it now only requires five clocks to output a final value (shorter latency).

Slide 5-17:

Configuring the Dataflow Channel



- Dataflow optimization
 - HLS implements a channel between submodules as either a **ping-pong** buffer or **FIFO** buffers
 - It places channels between the blocks to maintain the data rate



- For arrays, the channels will include memory elements to buffer the samples
- For scalars, the channel is a register with handshakes
- Dataflow optimization therefore has an area overhead
 - Additional memory blocks are added to the design

Slide 5-18:

FIFO vs. Ping-Pong RAM

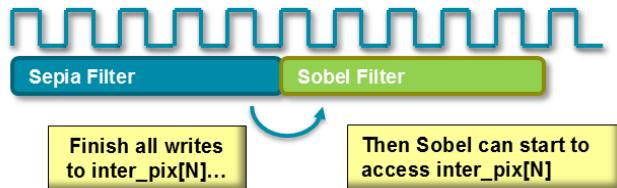


- Arrays are passed as single entities by default

```
//This memory is turned into a FIFO during optimization
rgb_pixel inter_pix[MAX_HEIGHT][MAX_WIDTH];

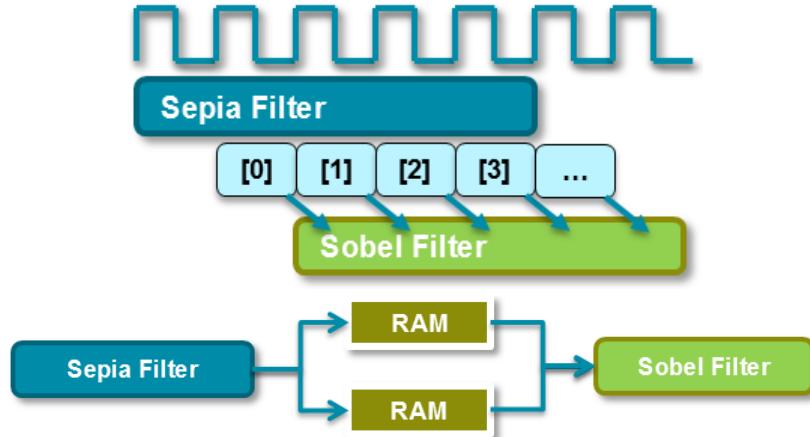
// Primary processing functions
sepia_filter(in_pix,inter_pix);
sobel_filter(inter_pix,out_pix2);
```

Sepia Filter
Sobel Filter



- Dataflow pipelining allows Sobel to start when data is ready

- Interval is improved
- Functions will operate in parallel



- Dataflow creates memory channels

- Created between loops or functions to store data samples
- Default is a ping pong buffer memory
 - Safe and reliable default
 - FIFO can also be used

Slide 5-19:

Dataflow Limitations (1)



- Must be single producer consumer; the following code violates the rule and dataflow does not work

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
  
    int templ[N];  
    Loop1: for(int i = 0; i < N; i++) {  
        templ[i] = data_in[i] * scale;  
    }  
    Loop2: for(int j = 0; j < N; j++) {  
        data_out1[j] = templ[j] * 123;  
    }  
    Loop3: for(int k = 0; k < N; k++) {  
        data_out2[j] = templ[k] * 456;  
    }  
}
```

- The fix

```
void Split (in[N], out1[N], out2[N]) {  
// Duplicated data  
    L1:for(int i=1;i<N;i++) {  
        out1[i] = in[i];  
        out2[i] = in[i];  
    }  
}  
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
  
    int templ[N], temp2[N]. temp3[N];  
    Loop1: for(int i = 0; i < N; i++) {  
        templ[i] = data_in[i] * scale;  
    }  
    Split(templ, temp2, temp3);  
    Loop2: for(int j = 0; j < N; j++) {  
        data_out1[j] = temp2[j] * 123;  
    }  
    Loop3: for(int k = 0; k < N; k++) {  
        data_out2[j] = temp3[k] * 456;  
    }  
}
```



For the dataflow optimization to work, the data must flow through the design from one task to the next. The following coding styles prevent the Vivado HLS tool from performing the dataflow optimization:

- Single producer consumer violations
- Bypassing tasks
- Feedback between tasks
- Conditional execution of tasks
- Loop scopes with variable bounds
- Loops with multiple exit conditions

For more information, refer to *Vivado Design Suite User Guide: High-Level Synthesis* (UG902).

Slide 5-20:

Dataflow Limitations (2)



- You cannot bypass a task; the following code violates this rule and dataflow does not work

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
  
    int temp1[N], temp2[N], temp3[N];  
    Loop1: for(int i = 0; i < N; i++) {  
        temp1[i] = data_in[i] * scale;  
        temp2[i] = data_in[i] >> scale;  
    }  
    Loop2: for(int j = 0; j < N; j++) {  
        temp3[j] = temp1[j] + 123;  
    }  
    Loop3: for(int k = 0; k < N; k++) {  
        data_out[j] = temp2[k] + temp3[k];  
    }  
}
```

- The fix: make it **systolic** like datapath

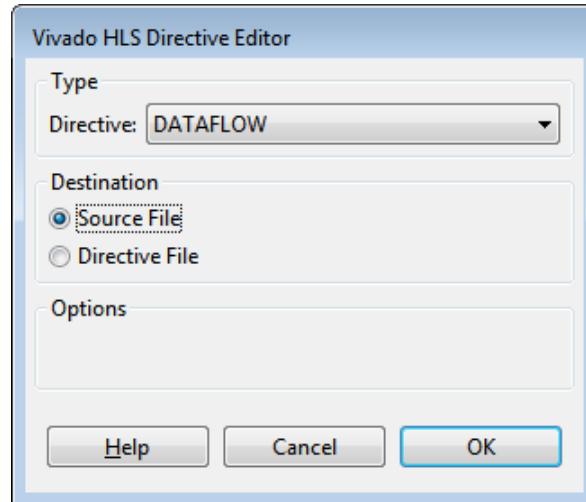
```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {  
  
    int temp1[N], temp2[N], temp3[N], temp4[N];  
    Loop1: for(int i = 0; i < N; i++) {  
        temp1[i] = data_in[i] * scale;  
        temp2[i] = data_in[i] >> scale;  
    }  
    Loop2: for(int j = 0; j < N; j++) {  
        temp3[j] = temp1[j] + 123;  
        temp4[j] = temp2[j];  
    }  
    Loop3: for(int k = 0; k < N; k++) {  
        data_out[j] = temp4[k] + temp3[k];  
    }  
}
```

Slide 5-21:

Dataflow Optimization Commands



- Dataflow is set using a directive
- Throughput rate
 - Throughput of the design will be defined by the maximum throughput of the functions or loops
 - Pipeline the loops and functions, then apply dataflow

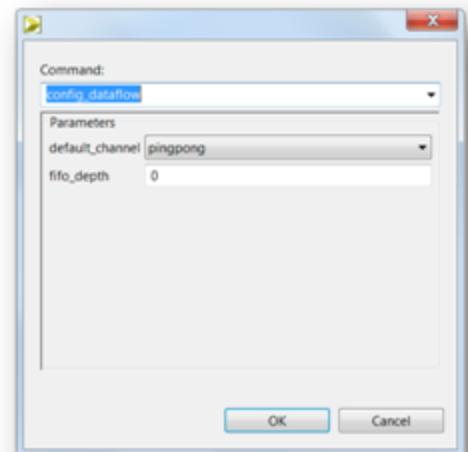


Slide 5-22:

Dataflow Optimization (1)



- Configuring dataflow memories
 - Between functions Vivado HLS tool uses ping-pong memory buffers by default
 - Memory size is defined by the maximum number of producer or consumer elements
 - Between loops Vivado HLS tool will determine if a FIFO can be used in place of a ping-pong buffer
 - Memories can be specified to be FIFOs using the dataflow configuration
 - Menu: Solution/Solution Settings/Configs
 - With FIFOs, can override the default size of the FIFO
 - **Note:** Setting the FIFO too small can result in an RTL verification failure



To use FIFO's the accesses must be sequential

FIFO Accesses:

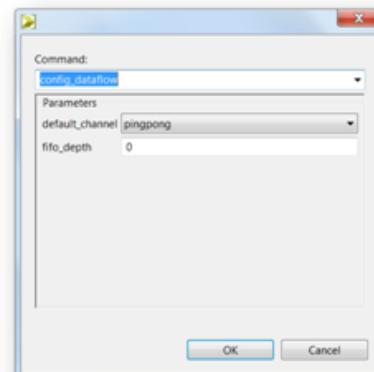
- If Vivado HLS can determine they are not in fact sequential, it will halt and issue a message.
- If it cannot confirm they are sequential it will issue a warning and proceed.

Slide 5-23:

Dataflow Optimization (2)



- Individual memory control
 - When the default is ping-pong
 - Select an array and mark it as streaming (directive **STREAM**) to implement the array as a FIFO
 - When the default is FIFO
- Select an array and mark it as streaming (directive **STREAM**) with option "**off**" to implement the array as a ping-pong



To use FIFO's the accesses must be sequential

FIFO Accesses:

- If Vivado HLS can determine they are **not** in fact sequential, it will halt and issue a message.
- If it cannot confirm they are sequential it will issue a warning and proceed.

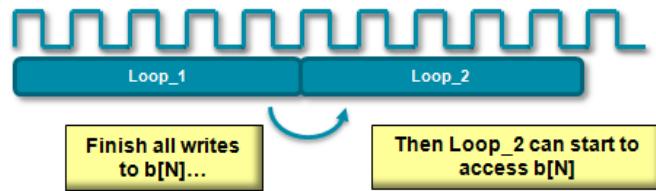
Slide 5-24:

Dataflow: Ideal for Streaming Arrays and Multi-Rate Functions (1)

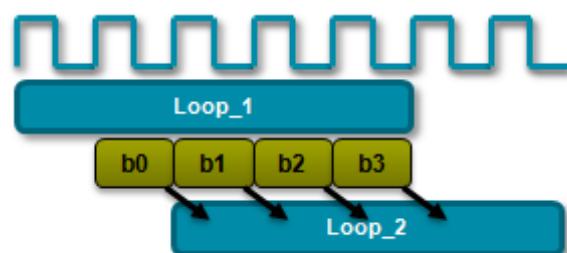


- Arrays are passed as single entities by default
 - This example uses loops but the same principle applies to functions

```
int a[N], b[N], c[N];
Loop_1: for (i=0;i<=N-1;i++) {
    b[i] = a[i] + in1;
}
Loop_2: for (i=0;i<=N-1;i++) {
    c[i] = b[i] * in2;
}
```



- Dataflow pipelining allows loop_2 to start when data is ready
 - Throughput is improved
 - Loops will operate in parallel
 - If dependencies allow



Slide 5-25:

Dataflow: Ideal for Streaming Arrays and Multi-Rate Functions (2)



- Multi-rate functions
 - Dataflow buffers data when one function or loop consumes or produces data a different rate from others
- I/O flow support
 - To take maximum advantage of dataflow in streaming designs, the I/O interfaces at both ends of the datapath should be streaming/handshake types (ap_hs or ap_fifo)

Slide 5-26:

Pipelining: Dataflow, Functions, and Loops



- Dataflow optimization
 - Dataflow optimization is "coarse grain" pipelining at the function and loop level
 - Increases concurrency between functions and loops
 - Only works on functions or loops at the top level of the hierarchy
 - Cannot be used in sub-functions
- Function and loop pipelining
 - "Fine grain" pipelining at the level of the operators (*, +, >>, etc.)
 - Allows the operations inside the function or loop to operate in parallel
 - Unrolls all sub-loops inside the function or loop being pipelined
 - Loops with variable bounds cannot be unrolled: this can prevent pipelining
 - Unrolling loops increases the number of operations and can increase memory and run time

Slide 5-27:

Summary

- Pipelining
 - Dataflow Optimization
 - **Summary**
- 

Slide 5-28:

Summary



- Use dataflow to make the functions or loops at the top level operate in parallel
 - Improves overall throughput of the design
- Pipeline the individual functions and loops to increase data throughput
 - Will improve the throughput of each sub-function or loop
 - Further improving the throughput
- Pipelining the entire design is an option
 - Requires every loop in the design to be unrolled
 - Can create lots of operators: impacts run time (like ungrouping all RTL)
 - May create high fanout nets: one FSM for the entire design
 - One large design with no hierarchy may be difficult to debug and analyze
- Look for bottlenecks
 - Often caused by array accesses

Capture Your Notes Here



Optimizing Structures for Performance

Slide 6-1:

2016.1

In this module, you will learn the performance limitations caused by arrays in the design and you also will learn some optimization techniques to handle arrays for improving performance. 30 minutes.

Slide 6-2:

Objectives

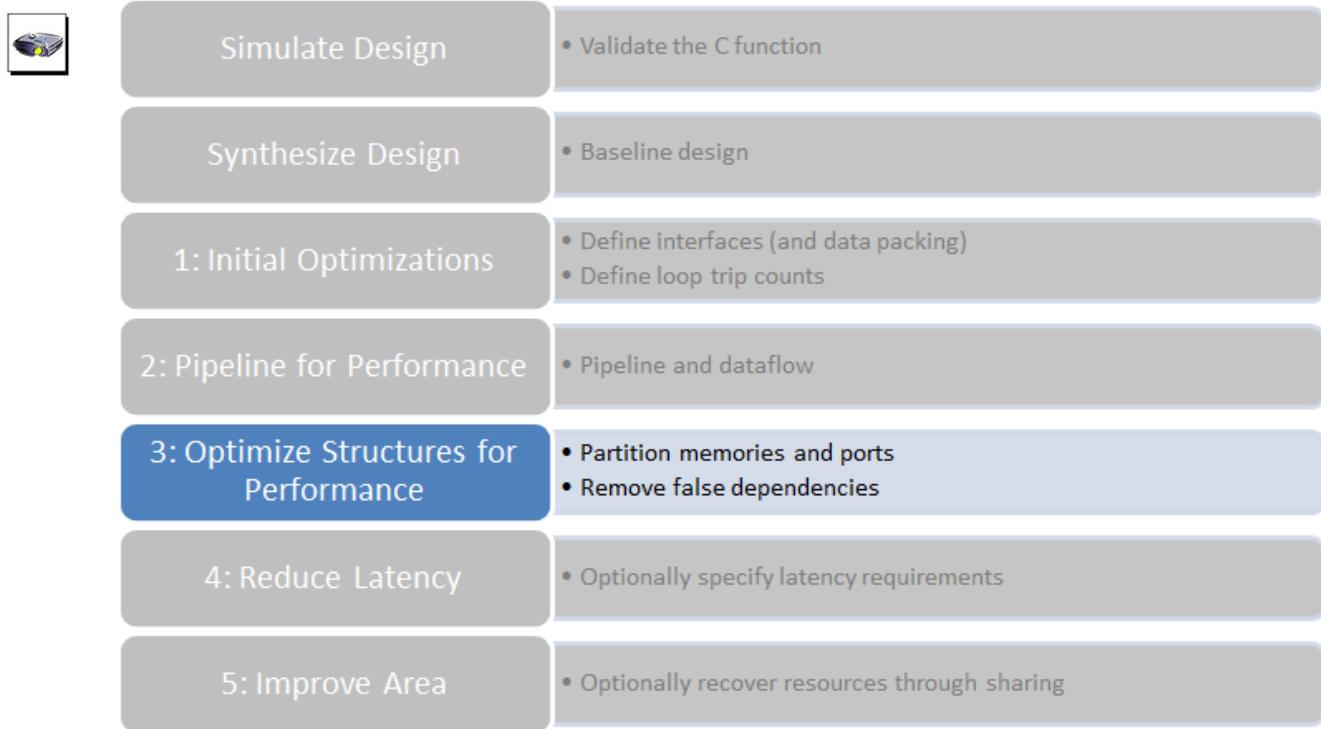


After completing this module, you will be able to:

- Describe arrays and their performance limitations in C
- Identify some of the techniques that optimize array performance

Slide 6-3:

HLS UltraFast Design Methodology



C code can contain descriptions that prevent a function or loop from being pipelined with the required performance. In some cases, this might require a code modification but in most cases these issues can be addressed by using other optimization directives.

Slide 6-4:

Step 3: Optimize Structures for Performance (1)



Directives and Configurations	Description
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers to improve access to data and remove block RAM bottlenecks.
DEPENDENCE	Used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
UNROLL	Unroll <i>for</i> loops to create multiple independent operations rather than a single collection of operations.

Slide 6-5:

Step 3: Optimize Structures for Performance (2)



Directives and Configurations	Description
Config Array Partition	Determines how arrays are partitioned, including global arrays and if the partitioning impacts array ports.
Config Compile	Controls synthesis-specific optimizations such as the automatic loop pipelining and floating point math optimizations.
Config Schedule	Determines the effort level to use during the synthesis scheduling phase and the verbosity of the output messages.

Slide 6-6:

Arrays in HLS



- **Arrays in HLS**
- Array Optimizations
- Summary

Slide 6-7:

Arrays: Performance Bottlenecks



- Arrays are intuitive and useful software constructs
 - Allow the C algorithm to be easily captured and understood
- Array accesses can often be performance bottlenecks
 - Arrays are targeted to a **default RAM** (dual port or single port)

```
void foo_top (...) {  
    ...  
    for (i = 2; i < N; i++)  
        mem[i] = mem[i-1]+mem[i-2];  
    }  
}
```



Even with a dual-port RAM, all reads and writes cannot be performed in one cycle

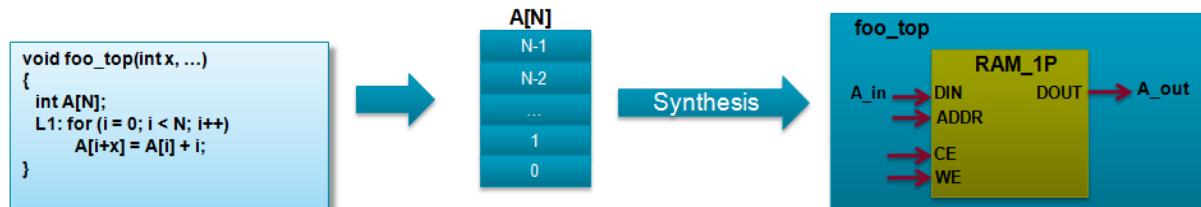
- Cannot pipeline with a throughput of 1
- Arrays can be partitioned and reshaped to produce higher data bandwidth
 - Allows more optimal configuration of the array
 - Provides better implementation of the memory resource

Slide 6-8:

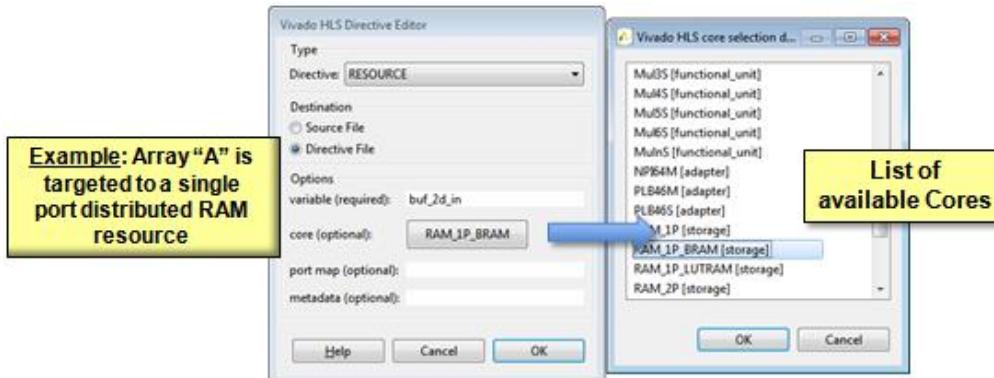
Arrays in HLS



- An array in C code is implemented by a memory in the RTL
 - By default, arrays are implemented as RAMs; optionally a FIFO



- Array can be targeted to any memory resource in the library
 - Ports and sequential operation are defined by the library model
 - All RAMs are listed in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)



Slide 6-9:

Array and RAM Selection



- If no RAM resource is selected
 - Vivado® HLS tool will determine the RAM to use
 - It will use a dual-port if it improves throughput
 - Else it will use a single-port
- Block RAM and LUTRAM selection
 - If none is made (e.g., resource RAM_1P used), RTL synthesis will determine if RAM is implemented as block RAM or LUTRAM
 - If the user specifies the RAM target (e.g., RAM_1P_BRAM or RAM_1P_LUTRAM is selected), Vivado HLS tool will obey the target
 - If LUTRAM is selected, Vivado HLS tool reports registers not block RAM

Slide 6-10:

Array Optimizations

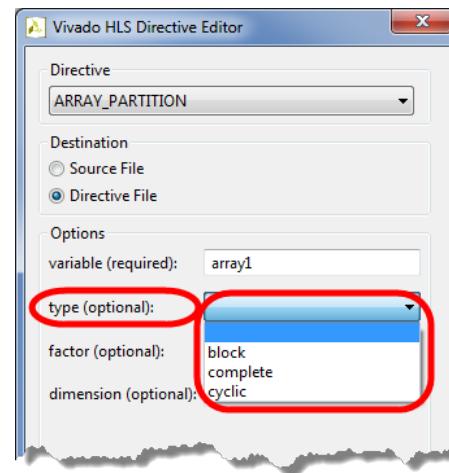
- Arrays in HLS
- ➡ ▪ **Array Optimizations**
- Summary

Slide 6-11:

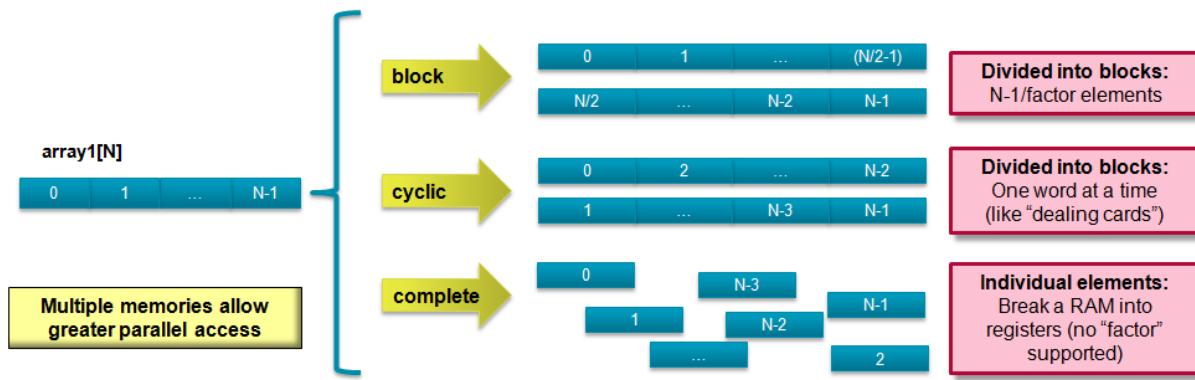
Array Partitioning



- Partitioning breaks an array into smaller elements
 - Arrays can be split along any dimension
 - All partitions inherit the same resource target



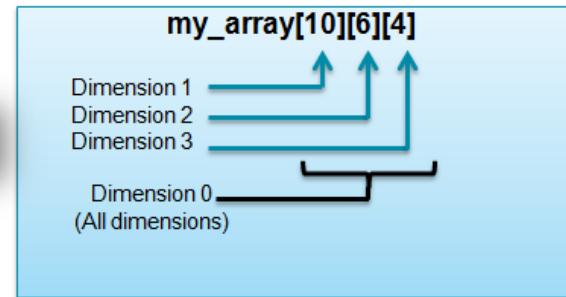
- Except of course "complete"



Slide 6-12:

Array Dimensions

- Array options can be performed on dimensions of the array



- Examples

my_array[10][6][4] → partition dimension 3 →

my_array_0[10][6]
my_array_1[10][6]
my_array_2[10][6]
my_array_3[10][6]

my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]

my_array[10][6][4] → partition dimension 1 →

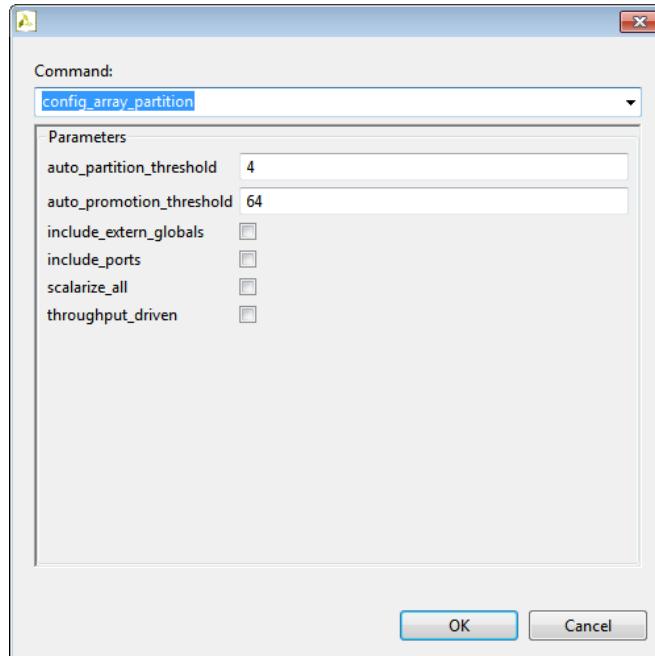
my_array[10][6][4] → partition dimension 0 → $10 \times 6 \times 4 = 240$ individual registers

Slide 6-13:

Configuring Array Partitioning (1)



- Vivado HLS tool can automatically partition arrays to improve throughput
 - Controlled via the array configuration command
- Auto-partition threshold
 - Arrays below the threshold are auto-partitioned
 - Unless the array has constant indexing
 - Index is not a variable (see next option)



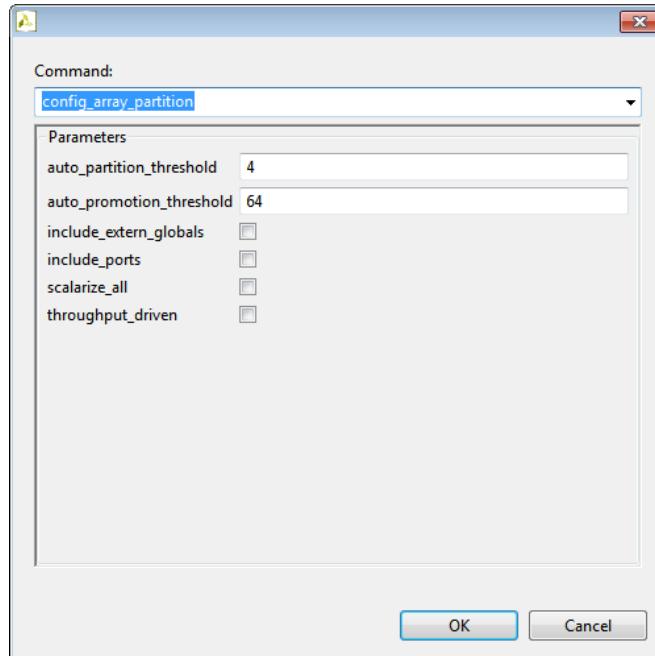
- Auto-promotion threshold
 - If the array has constant indexing
 - Array index is not a variable
 - Arrays above this threshold are promoted to memories
 - Arrays below this threshold are partitioned

Slide 6-14:

Configuring Array Partitioning (2)



- Include all arrays in partitioning
 - include_extern_globals and include_ports options will include any arrays defined in the global scope and on the /IO interface when partitioning is performed
 - Partitioning I/O arrays will result in multiple ports and change the interface
 - However, this may improve throughput
 - scalarize_all will ensure that all arrays are partitioned completely



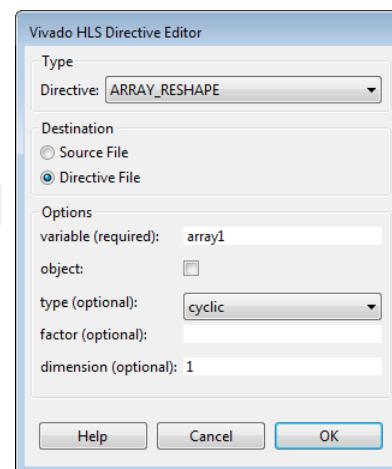
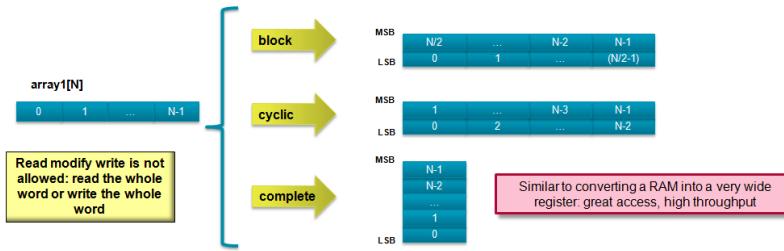
- Performance drives partitioning
- **throughput_driven** option will partition arrays automatically, *if it improves throughput*

Slide 6-15:

Array Reshaping



- Reshaping recombines partitioned arrays back into a single array



Slide 6-16:

Reshaping vs. Partitioning



- Both are useful for increasing the memory/data bandwidth
- Reshaping
 - Simply increases the width of the data word
 - Does not increase the number of memory ports
- Partitioning
 - Increases the memory ports; thus more I/O to deal with
 - Use it only if you have to use independent addressing
- Common error message: cue to use reshaping or partitioning

```
INFO: [SCHED 61] Pipelining loop 'SUM_LOOP'.
```

```
WARNING: [SCHED 69] Unable to schedule 'load' operation ('mem_load_2', bottleneck.c:62) on  
array 'mem' due to limited memory ports.
```

```
INFO: [SCHED 61] Pipelining result: Target II: 1, Final II: 2, Depth: 3.
```

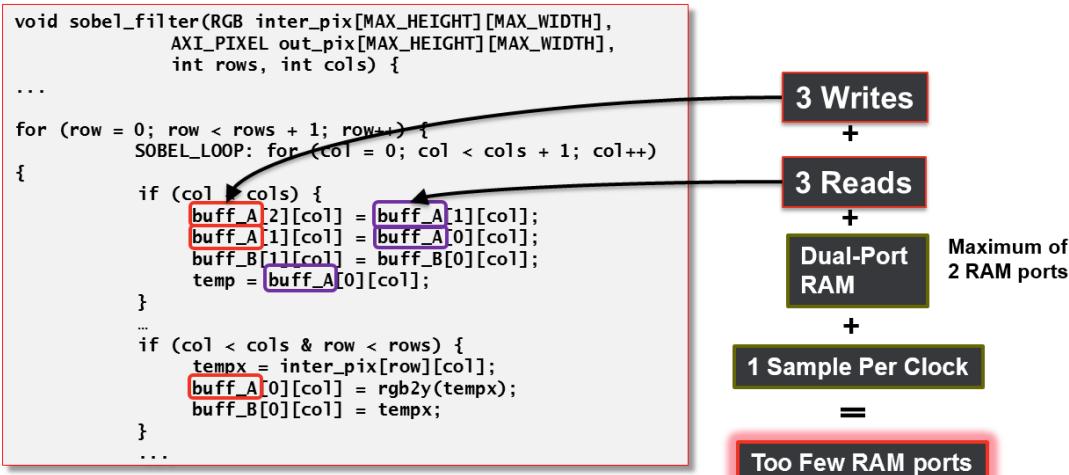
Slide 6-17:

Bandwidth Issues



- Array accesses (block RAM) can be bottlenecks inside functions or loops
 - Still prevents a II of 1 despite PIPELINE and DATAFLOW
 - Prevents processing of one sample per clock

```
@I [SCHED-61] Pipelining loop 'SOBEL_LOOP'.
@W [SCHED-69] Unable to schedule 'store' operation (image_demo.cpp:172) of
variable 'y' on array 'buff_A' due to limited resources (II = 2).
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 3, Depth: 13.
```

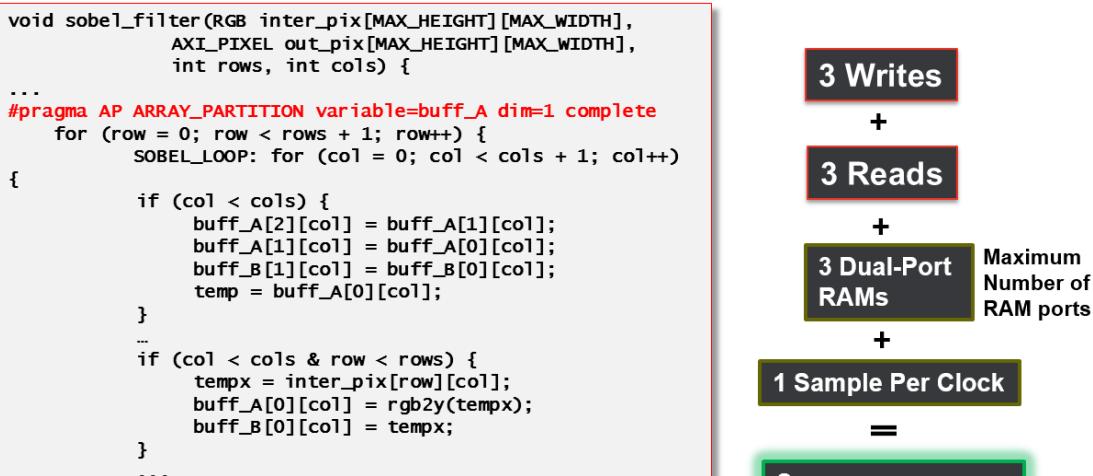


Slide 6-18:

Solution: Partitioning Arrays



- RAMs can create bottlenecks inside a function or loop
 - Partitioning improves availability of data
 - Similar for a sub-function, it may have too few ports

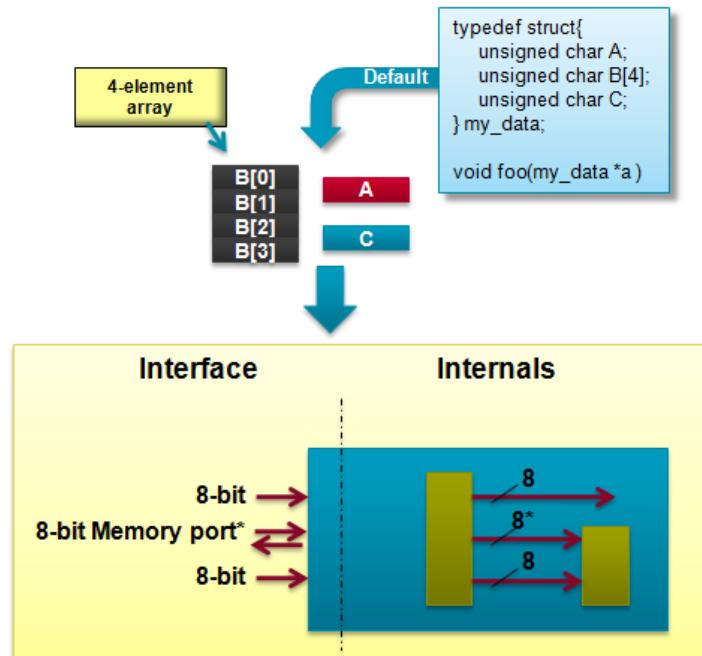


Slide 6-19:

Structs and Arrays: Default Handling



- Structs are a commonly used coding construct
- By default, a separate port is created for each element in a struct
- Internally
 - Separate buses and wires
 - Separate control logic, which may be more complex and slower and may increase latency
- Use the DATA_PACK directive to group them into a single element



* Assumes no array partitioning

Slide 6-20:

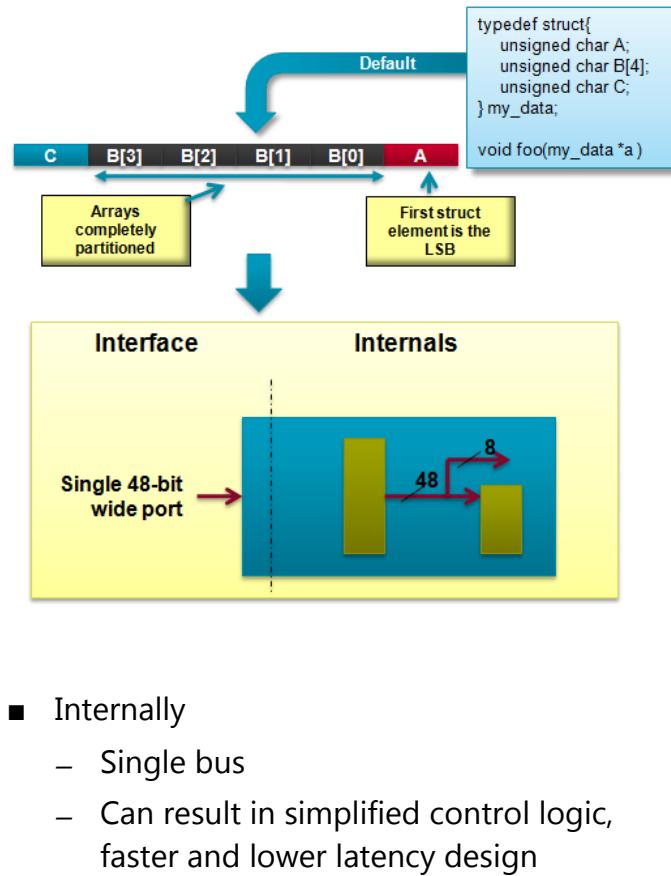
Data Packing



- Data packing groups structs internally and at the IO Interface
 - Creates a single wide bus of all struct elements

- Grouped structure
 - First element in the struct becomes the LSB
 - Last struct element becomes the MSB
 - Arrays are partitioning completely

- Less clutter to deal with
 - This means single port



- Internally
 - Single bus
 - Can result in simplified control logic, faster and lower latency design

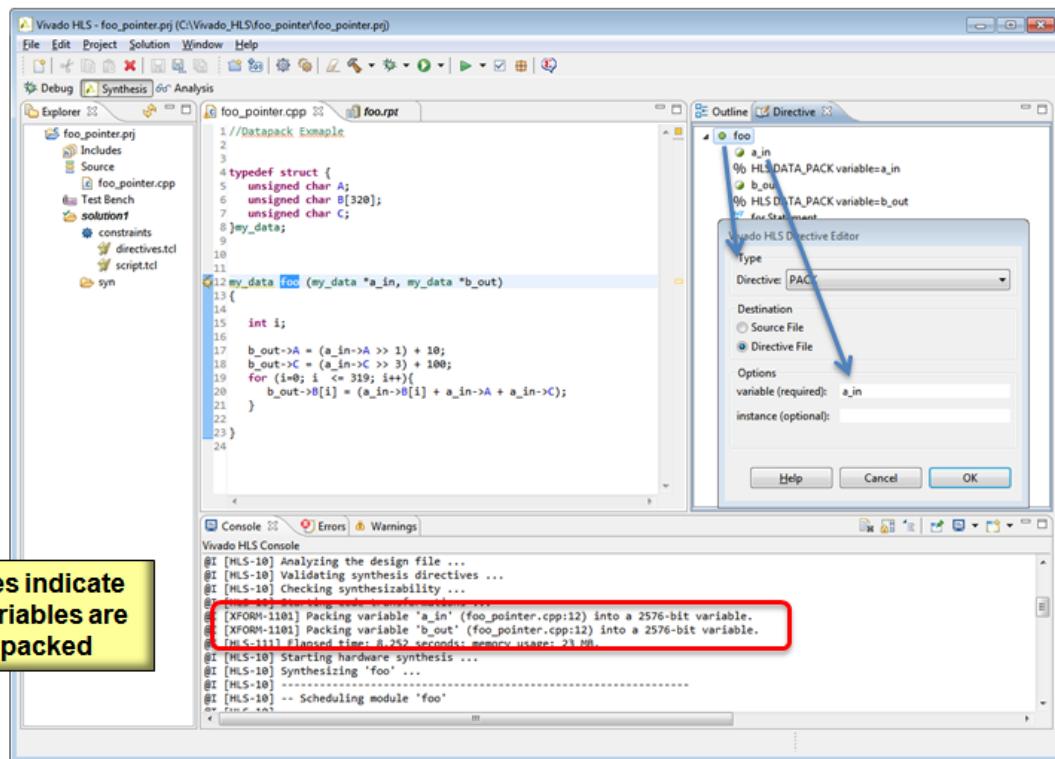
Slide 6-21:

Using Data Packing

- Apply the DATA_PACK directive
 - Select the function and specify the struct variable to pack



Messages indicate which variables are being packed



Slide 6-22:

Data Dependencies

- Design works fine if not pipelined



```

for (row = 0; row <= rows + 1; row++) {
    L2: for (col = 0; col <= cols + 1; col++) {
        #pragma HLS PIPELINE II=1
        if (col < cols) {
            buff_A[2][col] = buff_A[1][col]; Read
buff_A[1][col] = buff_A[0][col];
Write buff_B[1][col] = buff_B[0][col];
            temp = buff_A[0][col];
        }
    }
}

```

Buff_A[1][col]

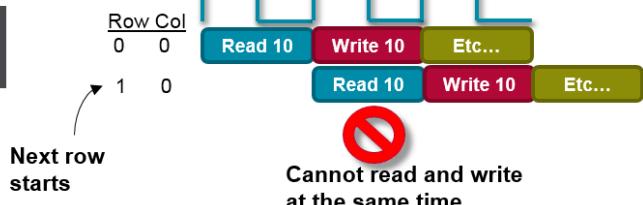
Intended Behavior



- Pipelined: implied dependency
 - Index limit "cols" is an input
 - What if cols=0, it means
 - There is only one iteration of loop L2

@W [SCHED-68] Unable to enforce a carried dependency constraint between 'store' operation ('image_demo.cpp:163') and 'load' operation ('buff_A_1_load', 'image_demo.cpp:162')

Implied Behavior if cols=0



Loop pipelining can be prevented by loop carry dependencies. Under certain complex scenarios, automatic dependence analysis can be too conservative and fail to filter out false dependencies.

In this example, the Vivado HLS tool does not have any knowledge about the value of `cols` and conservatively assumes that there is always a dependence between the write to `buff_A[1][col]` and the read from `buff_A[1][col]`.

Slide 6-23:

Specifying Dependencies



- Use the DEPENDENCE directive around the dependencies
 - Allows designers to inform tool of "external" conditions
 - There are no dependencies between loop iterations

```
for (row = 0; row <= rows + 1; row++) {  
    L2: for (col = 0; col <= cols + 1; col++) {  
        #pragma HLS PIPELINE II=1  
        #pragma AP dependence variable=buf_A inter false  
        #pragma AP dependence variable=buf_B inter false  
        if (col < cols) {  
            buf_A[2][col] = buf_A[1][col];  
            buf_A[1][col] = buf_A[0][col];  
            buf_B[1][col] = buf_B[0][col];  
            temp = buf_A[0][col];  
        }  
    }  
}
```



In an algorithm such as this, it is unlikely `cols` will ever be zero but the Vivado HLS tool cannot make assumptions about data dependencies. To overcome this deficiency, you can use the DEPENDENCE directive to provide the Vivado HLS tool with additional information about the dependencies. In this case, there is no dependence between loop iterations (in this case, for both `buf_A` and `buf_B`).

Note: Specifying a false dependency, when in fact the dependency is not false, can result in incorrect hardware. Be sure that dependencies are correct (true or false) before specifying them.

When specifying dependencies there are two main types:

- Inter: Specifies that the dependency is between different iterations of the same loop. If this is specified as false, it allows the Vivado HLS tool to perform operations in parallel if the pipelined or loop is unrolled or partially unrolled and prevents such concurrent operation when specified as true.
- Intra: Specifies dependence within the same iteration of a loop—for example an array being accessed at the start and end of the same iteration.

Data dependencies are a much harder issues to resolve and often require change to the source code.

Slide 6-24:

Summary

- Arrays in HLS
 - Array Optimizations
 - **Summary**
- 

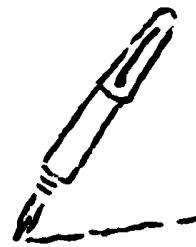
Slide 6-25:

Summary



- Vivado HLS tool will determine which type of RAM to implement depending on the types of access and requirements
 - Use the Resource directive to explicitly state which RAM to use
- Arrays: performance limiting
 - Arrays can create performance bottleneck if not handled properly
 - Use Array Partitioning, Reshaping, and Data packing directives to achieve throughput

Capture Your Notes Here



Reducing Latency

Slide 7-1:

2016.1

This module describes how to optimize the C design to improve latency. You will also learn how to apply directives and create and work with multiple solutions in the Vivado® HLS tool. 45 minutes.

Slide 7-2:

Objectives

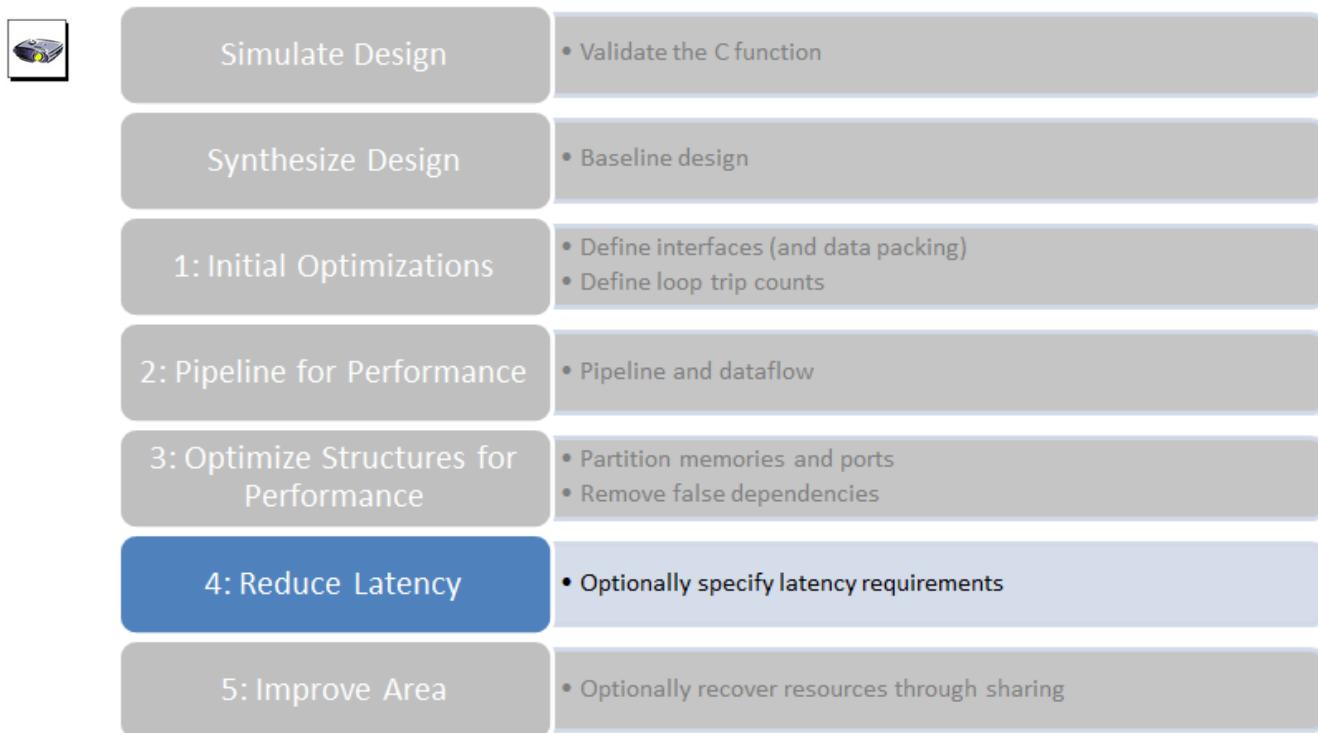


After completing this module, you will be able to:

- Describe default implementations of loops
- Handle loops to achieve the desired latency and throughput

Slide 7-3:

HLS UltraFast Design Methodology



Slide 7-4:

Step 4: Reduce Latency

Directives and Configurations	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.



When the Vivado HLS tool finishes minimizing the initiation interval, it automatically seeks to minimize the latency. The optimization directives listed in the table above can help reduce or specify a particular latency.

These are generally not required when the loops and function are *pipelined* as in most applications the latency is not critical; throughput typically is.

If the loops and functions are not pipelined, the throughput will be limited by the latency, because the task will not start reading the next set of inputs until the prior task has completed.

The LATENCY directive is used to specify the required latency. The loop optimization directives can be used to flatten a loop hierarchy or merge serial loops together. The benefit to the latency is due to the fact that it typically costs a clock cycle to enter and leave a loop. The fewer the number of transitions between loops, the fewer number of clock cycles a design will take to complete.

Slide 7-5:

Improving Latency



- **Improving Latency**
- Loops: Impact on Latency
- Summary

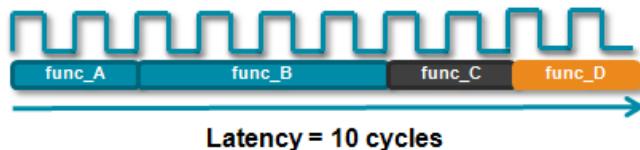
Slide 7-6:

Review: Latency and Throughput

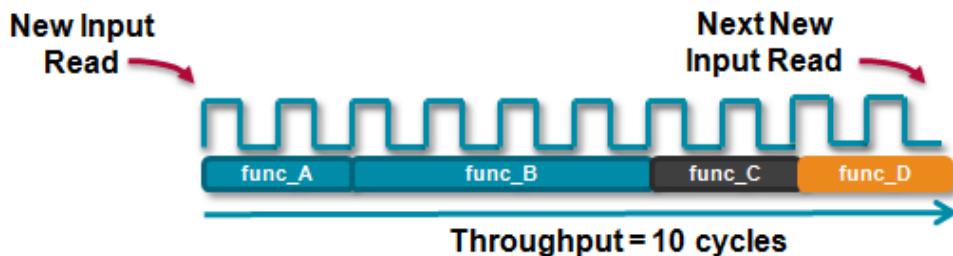


- Design latency
 - Latency of the design is the number of cycles it takes to output the result
 - In this example the latency is 10 cycles

```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(...);
    func_B(...);
    func_C(...);
    func_D(...);
    return res;
}
```



- Design throughput
 - Throughput of the design is the number of cycles between new inputs
 - By default (no concurrency) this is the same as latency
 - Next start/read happens when this transaction ends

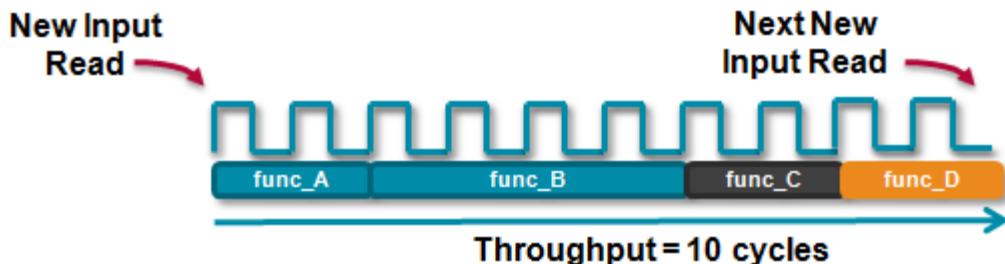


Slide 7-7:

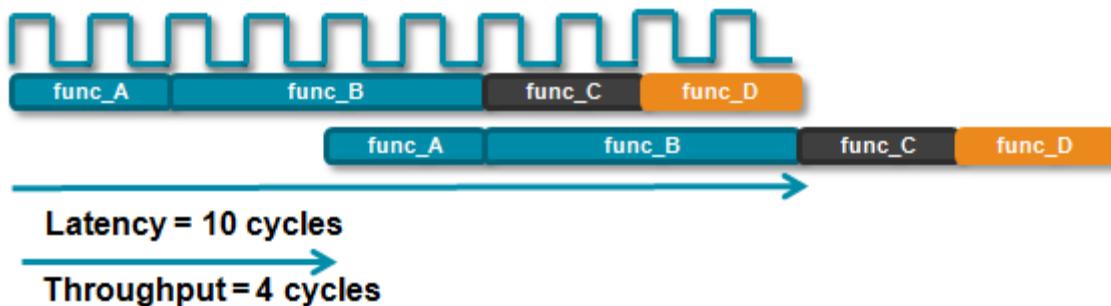
Latency and Throughput



- In the absence of any concurrency
 - Latency is the same as throughput



- Pipelining for higher throughput
 - Vivado HLS tool can pipeline functions and loops to improve throughput



- Latency and throughput are related

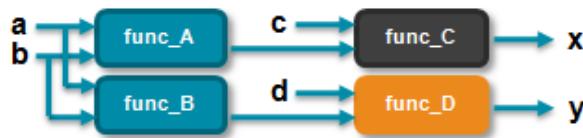
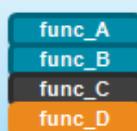
Slide 7-8:

Vivado HLS Tool: Minimize Latency (1)



- Vivado HLS tool will minimize latency by default
 - Throughput is prioritized above latency (no throughput directive is specified here)
 - In this example
 - Functions are connected as shown below
 - Function B takes longer than any other functions

```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(a,b,&t1);
    func_B(a,b,&t2);
    func_C(c,t1,&x)
    func_D(d,t2,&y)
}
```

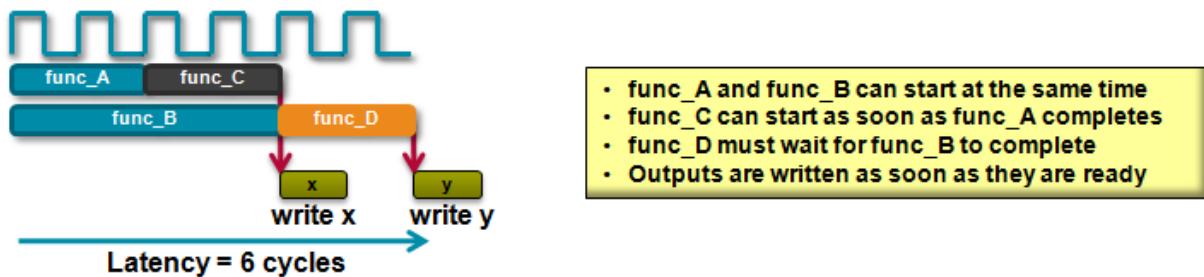


Slide 7-9:

Vivado HLS Tool: Minimize Latency (2)



- Vivado HLS tool will automatically take advantage of the parallelism
 - Will schedule functions to start as soon as they can
 - **Note:** it will not do this for loops within a function; by default they are executed in sequence



Slide 7-10:

Vivado HLS Tool Default Behavior: Minimizing Latency



- Functions
 - Vivado HLS tool will seek to minimize latency by allowing functions to operate in parallel
 - As shown in the previous slide
- Loops
 - Vivado HLS tool will not schedule loops to operate in parallel by default
 - Dataflow optimization must be used or the loops must be unrolled
 - Both techniques are discussed in detail later
- Operations
 - Vivado HLS tool will seek to minimize latency by allowing the operations to occur in parallel
 - It does this within functions **and** within loops

```
Loop:for(i=1;i<3;i++) {
    op_Read;
    op_Compute;
    op_Write;
}
```



```
void foo(...) {
    op_Read;
    op_Compute;
    op_Write;
}
```

Example with Sequential Operations



Example of Minimizing Latency with Parallel Operations

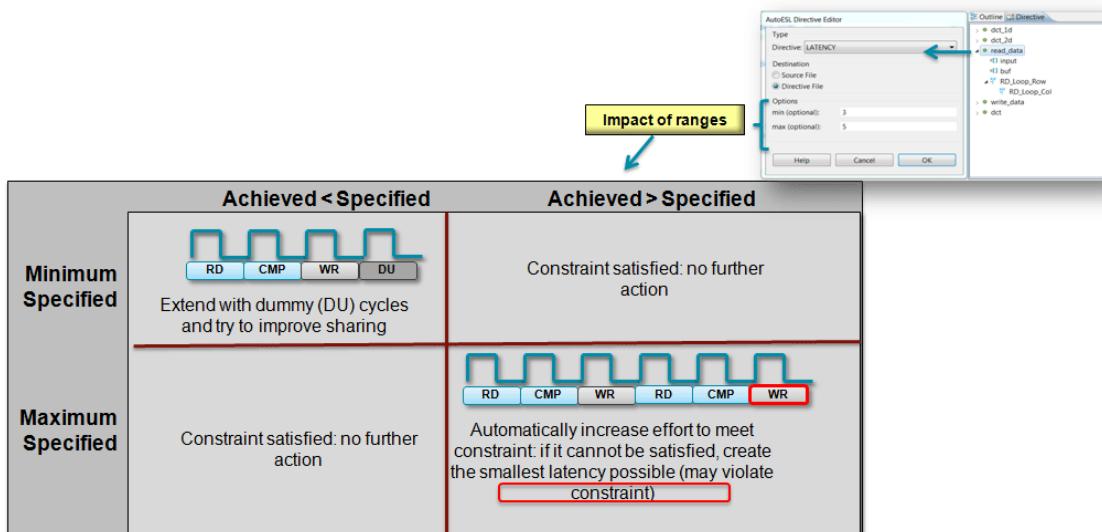


Slide 7-11:

Latency Constraints



- Latency constraints can be specified
 - Can define a minimum and/or maximum latency for the location
 - Applied to all objects in the specified scope
 - No range specification: schedule for minimum
 - Which is the default

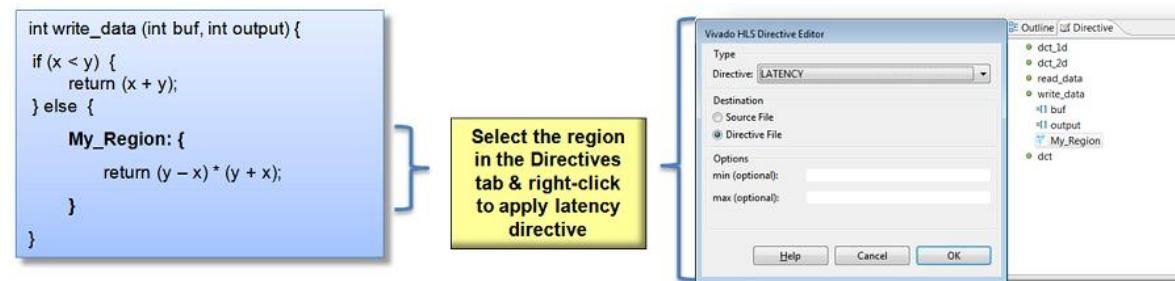


Slide 7-12:

Regions: Specific Latency Constraints



- Latency directives can be applied on functions, loops, and regions
- Use regions to specify *specific* locations for latency constraints
 - Region is any set of named braces {...a region...}
 - The region **My_Region** is shown in this example
 - Allows the constraint to be applied to a specific range of code
 - Here, only the else branch has a latency constraint



Slide 7-13:

Loops: Impact on Latency



- Improving Latency
- Loops: Impact on Latency
- Summary

Slide 7-14:

Loops: Basics

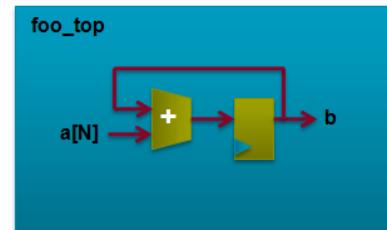


- By default, loops are rolled
 - Each C loop iteration → implemented in the same state
 - Each C loop iteration → implemented with same resources

```
void foo_top (...) {  
    ...  
    Add: for (i=3;i>=0;i--) {  
        b = a[i] + b;  
    }  
    ...
```

Loops require labels if they are to be
referenced by Tcl directives
(GUI will auto-add labels)

Synthesis



- Loops can be unrolled if their indices are statically determinable at elaboration time
 - Not when the number of iterations is variable

Slide 7-15:

Rolled Loops



- Loop iteration runs on the single hardware resource
 - Accumulation in a loop is one adder, for example
- Loops imply latency
- Incrementing a loop counter always consumes at least one clock cycle

```
void foo (...) {
...
add: for (i=0;i<=3;i++)
{
    b = a[i] + b;
...
}
```



Example: This loop (without directives) will always take at least four clock cycles

Slide 7-16:

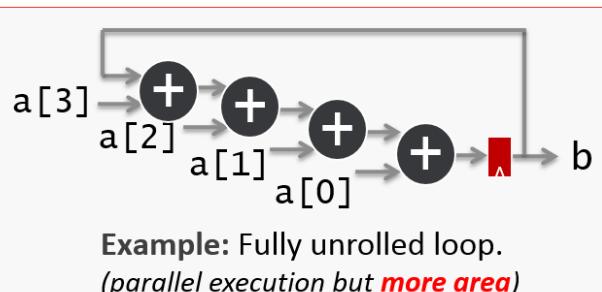
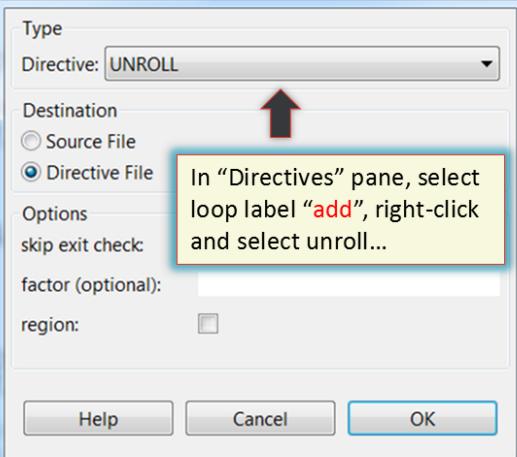
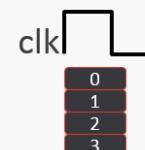
Unrolled Loops



```
void foo (...) {
...
add: for (i=0;i<=3;i++)
{
    b = a[i] + b;
...
}
```

Default: 4 cycles

Unroll: 1 cycle



Slide 7-17:

Partial Unrolling (1)



- Fully unrolling loops can create a lot of hardware
- Loops can be partially unrolled
 - Provides the type of exploration shown in the previous slide
- Unrolling by itself does not help increase throughput in some cases when data access is limited
- **Note:** Most likely you will use pipelining more often than unroll

Slide 7-18:

Partial Unrolling (2)



- Partial unrolling
 - A standard loop of N iterations can be unrolled to by a factor
 - For example **unroll by a factor 2** to have $N/2$ iterations
 - Similar to writing new code as shown on the right
 - The break accounts for the condition when $N/2$ is not an integer
 - If “ i ” is known to be an integer multiple of N
 - Can **remove the exit check** (and associated logic)
 - Vivado HLS tool is not always be able to determine this is true (e.g., if N is an input argument)
 - User takes responsibility: verify!

```
Add: for(int i = 0; i < N; i++) {  
    a[i] = b[i] + c[i];  
}
```

```
Add: for(int i = 0; i < N; i += 2) {  
    a[i] = b[i] + c[i];  
    if (i+1 >= N) break;  
    a[i+1] = b[i+1] + c[i+1];  
}
```

**Effective code
after compiler
transformation**

```
for(int i = 0; i < N; i += 2) {  
    a[i] = b[i] + c[i];  
    a[i+1] = b[i+1] + c[i+1];  
}
```

**An extra adder
for $N/2$ cycles
trade-off**

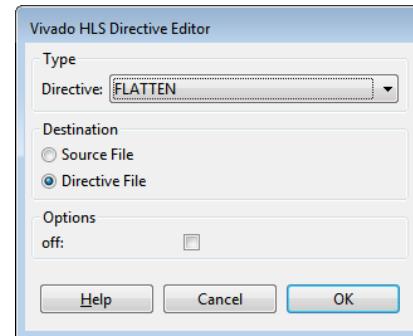
There is an adder for the loop iteration count “ i ” itself.

Slide 7-19:

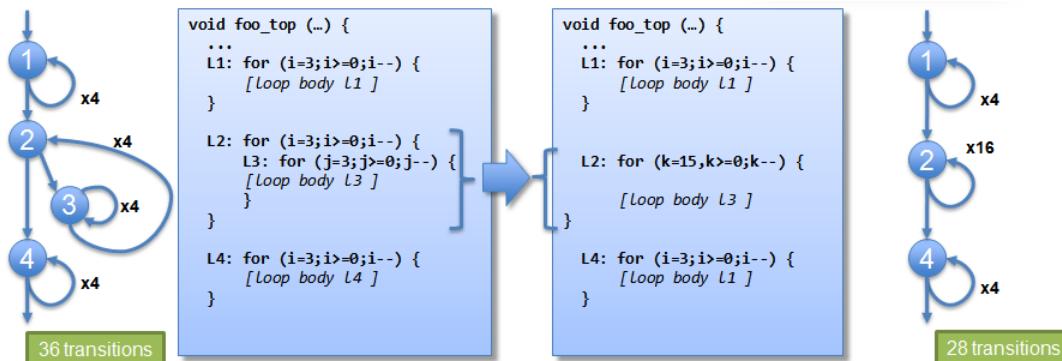
Loop Flattening



- Vivado HLS tool can automatically flatten **nested** loops
 - A faster approach than manually changing the code
- Flattening should be specified on the **inner-most loop**
 - It will be flattened into the loop above
 - The **off** option can prevent loops in the hierarchy from being flattened



- Loops will be **flattened by default**: select *off* to disable



Slide 7-20:

Converting Imperfect Loop to a Perfect Loop



- Fewer the number of loop levels, the better for latency
- If more than one loop level, try to make it into a perfect loop

```

94 void loop_imperfect(din_t A[N], dout_t B[N]) {
95
96     int i,j;
97     dint_t acc;
98
99     LOOP_I:for(i=0; i < 20; i++){
100         acc = 0;
101         LOOP_J: for(j=0; j < 20; j++){
102             acc += A[i] * j;
103         }
104         B[i] = acc / 20;
105     }
106 }
107 }

94 void loop_perfect(din_t A[N], dout_t B[N]) {
95
96     int i,j;
97     dint_t acc;
98
99     LOOP_I:for(i=0; i < 20; i++){
100         LOOP_J: for(j=0; i < 20; j++){
101             if(j==0) acc = 0;
102             acc += A[i] * j;
103             if(j==19) B[i] = acc / 20;
104         }
105     }
106 }
107 }

```

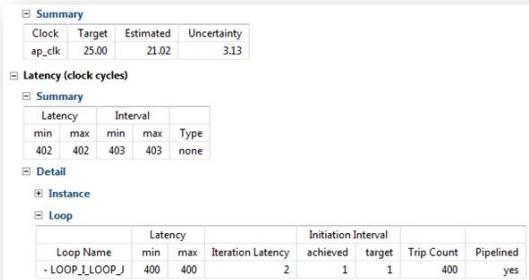
Slide 7-21:

Perfect and Semi-Perfect Loops (1)

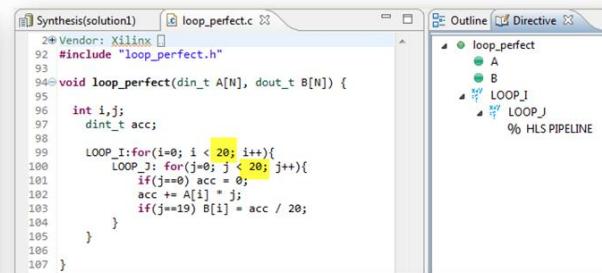


- Perfect and semi-perfect loop
 - Can be **flattened**
 - Sometimes imperfect loop can be flattened too

- Perfect loop
 - Only the inner-most loop has body (contents)
 - No logic specified between the loop statements
 - Loop bounds are **constant**



```
Loop_outer: for ( i=3;i>=0;i-- ) {
    Loop_inner: for ( j=3;j>=0;j-- ) {
        [loop body]
    }
}
```



Slide 7-22:

Perfect and Semi-Perfect Loops (2)



- Semi-perfect loops
 - Only the inner-most loop has body (contents)
 - No logic specified between the loop statements
 - Outer-most loop bound can be **variable**

- Imperfect loop
 - Check how the loops are flattened

```
Loop_outer: for ( i=3;i>N;i-- ) {
    Loop_inner: for ( j=3;j>=0;j-- ) {
        [loop body]
    }
}
```

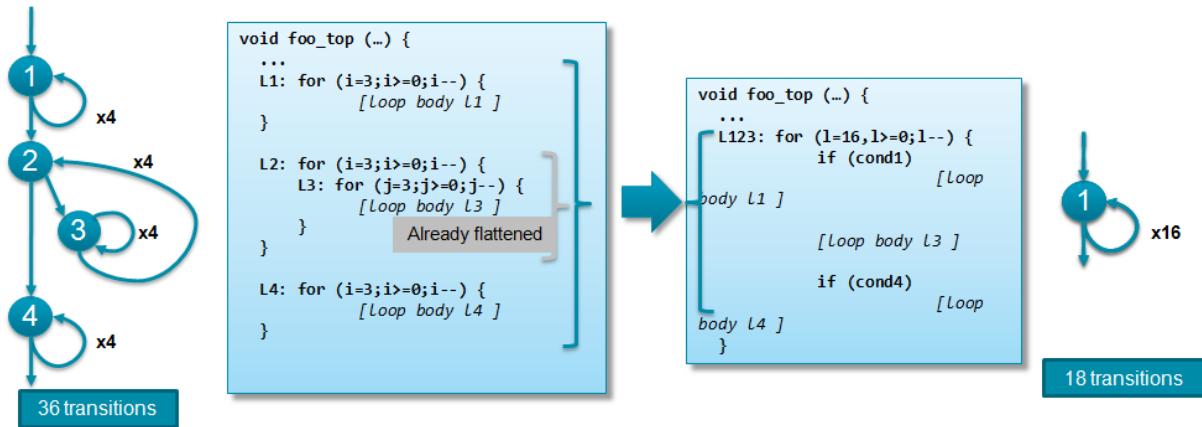
```
Loop_outer: for ( i=3;i>N;i-- ) {
    [loop body]
    Loop_inner: for ( j=3;j>=M;j-- ) {
        [loop body]
    }
}
```

Slide 7-23:

Loop Merging



- Vivado HLS tool can automatically merge loops
 - A faster approach than manually changing the code
 - Allows for more efficient architecture explorations
 - **FIFO reads, which must occur in strict order, can prevent loop merging**
 - Can be done with the **force** option: user takes responsibility for correctness

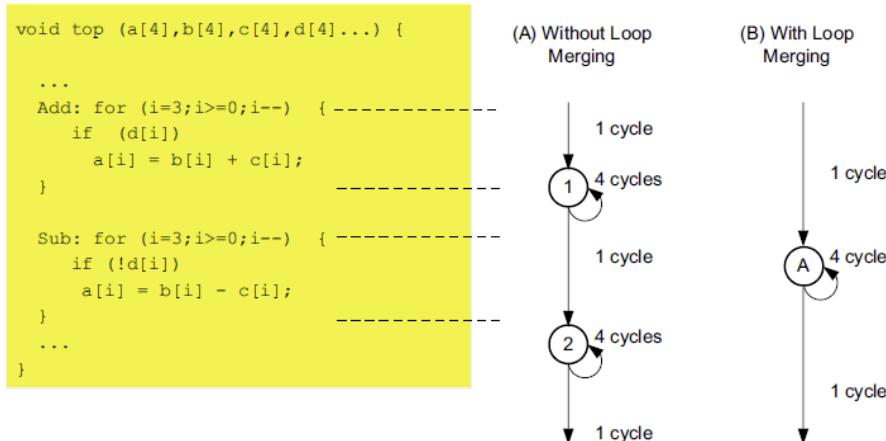


Slide 7-24:

Loop Merge Rules



- If loop bound are all variables, they must have the same value
- If loop bounds are constants, maximum constant value is used as bound of merged loop
- Loops with both variable bound and constant bound cannot be merged
- Code between loops to be merged cannot have side effects: multiple execution of this code should generate the same results
 - $a=b$ is allowed; $a=a+1$ is not
- Loops cannot be merged when they contain FIFO accesses
 - Merging would change the order of reads and writes from a FIFO; these must always occur in sequence



In the figure above, (A) shows how, by default, each rolled loop in the design creates at least one state in the FSM. Moving between these states costs clock cycles.

Assuming each loop iteration requires one clock cycle, it takes a total of 11 cycles to execute both loops:

- One clock cycle to enter the ADD loop.
- Four clock cycles to execute the ADD loop.
- One clock cycle to exit ADD and enter SUB.
- Four clock cycles to execute the SUB loop.
- One clock cycle to exit the SUB loop.

In this simple example it is obvious that an else branch in the ADD loop would also solve the issue but in a more complex example it may be less obvious and the more intuitive coding style may have greater advantages.

The LOOP_MERGE optimization directive is used to automatically merge loops. The LOOP_MERGE directive will seek to merge all loops within the scope it is placed.

In the above example, merging the loops creates a control structure similar to that shown in (B) in the figure, which requires only six clocks to complete.

Slide 7-25:

Loop Reports



- Vivado HLS tool reports the latency of loops
 - Shown in the report file and GUI
- Given a **variable loop** index, the latency cannot be reported
 - Vivado HLS tool does not know the limits of the loop index
 - Therefore, latency reports show unknown values
- **Loop trip** count (iteration count) can be specified
 - Apply to the loop in the directives pane
 - Allows the reports to show an estimated latency
- Other option is to use "**assert**"

The screenshot shows the Vivado HLS Directive Editor and a Performance Estimates report side-by-side. The Directive Editor window has 'LOOP_TRIPCOUNT' selected in the 'Directive' dropdown. Under 'Options', 'min (optional)' is set to 200 and 'max (optional)' is set to 1920. The Performance Estimates report shows a summary table for latency and interval, with the 'Loop' section expanded to show details for the loop being analyzed.

```

void foo(in1[N][M], in2[N][M], Rows, Cols...) {
    ...
    L1:for(i=1;i<Rows;i++) {
        #pragma HLS loop_tripcount min=1080 max=1080 avg=1080
        L2:for(j=0;j<Cols;j++) {
            #pragma HLS loop_tripcount min=1920 max=1920 avg=1920
            #pragma HLS PIPELINE
                out[i][j] = in1[i][j] + in2[i][j];
        }
    }
}
  
```

Impacts Reporting – Not Synthesis



If loops have variable bounds, the Vivado HLS tool cannot determine the number of loop iterations; hence, it cannot report the total latency of loops on the design. This results in '?' in the report for loop latencies.

You can provide a minimum, typical, or maximum value for the loop iterations by using the TRIPCOUNT directive. The tool can use this directive to generate meaningful reports. The TRIPCOUNT directive impacts reporting only and does not impact synthesis.

Slide 7-26:

Loops with Variable Limits: Assertion Support



- Assertions are supported for synthesis
 - Can be used to define bit widths for synthesis
 - C code asserts the **maximum limit** of the loop bound
 - Replaces the need for a TRIPCOUNT directive

Without Assertions

```
SUM_X:for (i=0;i<=xlimit; i++) {
    X_accum += A[i];
    X[i] = X_accum;
}

SUM_Y:for (i=0;i<=ylimit; i++) {
    Y_accum += B[i];
    Y[i] = Y_accum;
}
```



Reported Loop Latency:			
	Target II	Trip Count	Pipelined
- SUM_X	1 ~ 256	no	
- SUM_Y	1 ~ 256	no	

With Assertions

```
assert(xlimit<32);
SUM_X:for (i=0;i<=xlimit; i++) {
    X_accum += A[i];
    X[i] = X_accum;
}
assert(ylimit<16);
SUM_Y:for (i=0;i<=ylimit; i++) {
    Y_accum += B[i];
    Y[i] = Y_accum;
}
```



Reported Loop Latency:			
	Target II	Trip Count	Pipelined
- SUM_X	1 ~ 32	no	
- SUM_Y	1 ~ 16	no	

Index counter hardware is accurately sized

Slide 7-27:

Summary

- Improving Latency
- Loops: Impact on Latency
- **Summary**



Slide 7-28:

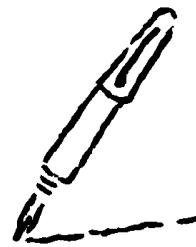
Summary



- Loop optimizations
 - Loops and loop transitions can imply cycles: watch them
 - Specify latency directives
 - Latency can be improved by minimizing the number of loop boundaries
 - Rolled loops (default) enforce sharing at the expense of latency
 - Entry and exits to loops costs clock cycles
 - Merge and flatten loops to reduce loop transition overheads

- Loops with variable limits
 - Assertions are supported for synthesis
 - Use TRIPCOUNT directive for reporting

Capture Your Notes Here



Improving Area

Slide 8-1:

2016.1

This module describes different methods for improving resource utilization. 30 minutes.

Slide 8-2:

Objectives

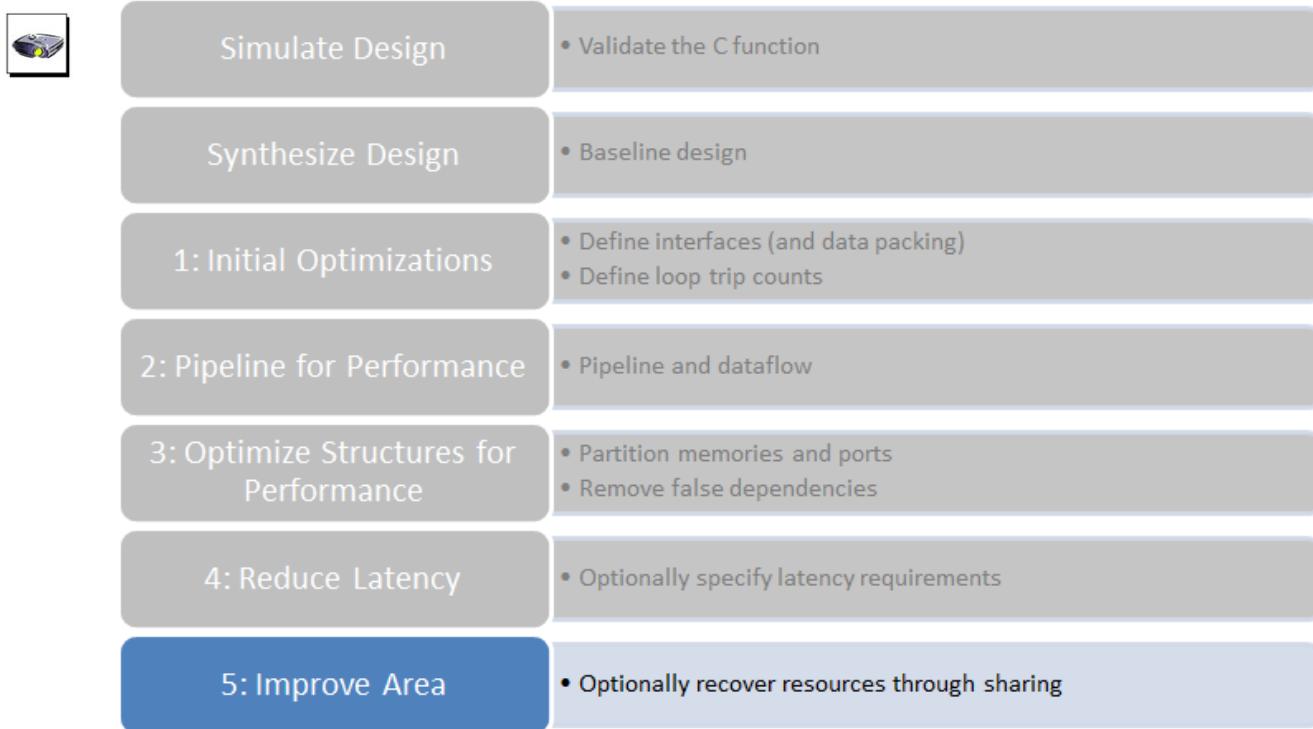


After completing this module, you will be able to:

- Describe different methods for improving resource utilization
- Control the structure of the design by using directives to improve area
- Explain how arbitrary precision types can result in optimal resource usage

Slide 8-3:

HLS UltraFast Design Methodology



After meeting the required performance target (or II), the next step is to reduce the area while maintaining the same performance.

If you used the DATAFLOW optimization and the Vivado® HLS tool cannot determine whether the tasks in the design are streaming data, the Vivado HLS tool implements the memory channels between dataflow tasks using ping-pong buffers.

If the design is pipelined and the data is streaming from one task to the next, you can greatly reduce the area by using the dataflow configuration **config_dataflow** to convert the ping-pong buffers used in the default memory channels into FIFO buffers. You can then set the FIFO depth to the minimum required size.

The dataflow configuration **config_dataflow** specifies the default implementation for all memory channels. You can use the STREAM directive to specify which individual arrays to implement as block RAM and which to implement as FIFOs.

If the design is implemented using an **hls::stream** I/O protocol, the memory channels default to FIFOs with a depth of 1, and the dataflow configuration is not required. However, the STREAM directive can be used to increase the size of the FIFOs for cases where a task outputs more data than it consumed, such as interpolation.

Slide 8-4:

Step 5: Improve Area (1)



Directives and Configurations	Description
ALLOCATION	Specify a limit for the number of operations, cores or functions used. This can force the sharing of hardware resources and might increase latency.
ARRAY_MAP	Combines multiple smaller arrays into a single large array to help reduce block RAM resources.
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.
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.
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.
RESOURCE	Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL.



The ALLOCATION and RESOURCE directives are used to limit the number of operations and to select which cores (or resources) are used to implement the operations. For example, you could limit the function or loop to using only one multiplier and specify it to be implemented using a pipelined multiplier. The binding configuration is used to globally limit the use of a particular operation.

If the ARRAY_PARTITION directive is used to improve the initiation interval, you might want to consider using the ARRAY_RESHAPE directive instead. The ARRAY_RESHAPE optimization performs a similar task to array partitioning; however the reshape optimization recombines the elements created by partitioning into a single block RAM with wider data ports.

If the C code contains a series of loops with similar indexing, merging the loops with the LOOP_MERGE directive might allow some optimizations to occur.

Finally, in cases where a section of code in a pipeline region is only required to operate at an initiation interval lower than the rest of the region, the OCCURRENCE directive is used to indicate this logic can be optimized to execute at a lower rate.

Slide 8-5:

Step 5: Improve Area (2)

 Directives and Configurations	Description
STREAM	Specifies that a specific memory channel is to be implemented as a FIFO or RAM during dataflow optimization.
Config Bind	Determines the effort level to use during the synthesis binding phase and can be used to globally minimize the number of operations used.
Config Dataflow	This configuration specifies the default memory channel and FIFO depth in dataflow optimization.

Slide 8-6:

Improving Area



- Control the number of elements
 - Directives can be used to control scheduling and binding
- Control the design hierarchy
 - Like RTL synthesis, removing the hierarchy can help optimize across function and loop boundaries
 - Functions can be inlined
 - Loops can be unrolled
- Array implementation
 - Vivado® HLS tool provides directives for combining memories
 - Allowing a single large memory to be used instead of multiple smaller memories
- Bit-width optimization
 - Arbitrary precision types ensure correct operator sizing

Slide 8-7:

Controlling the Resources Used



Controlling the Resources Used

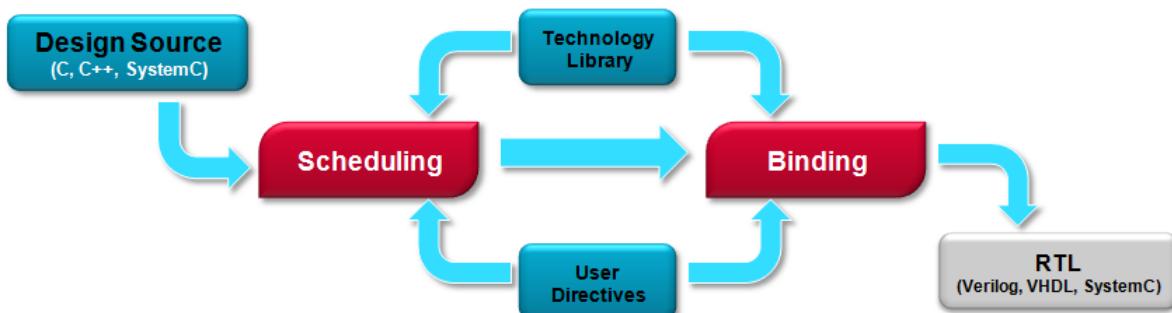
- Controlling the Structure of the Design
- Importance of Arbitrary Precision Types
- Summary

Slide 8-8:

Review: Control Scheduling and Binding



- Scheduling and binding
 - Scheduling and binding are the processes at the heart of HLS



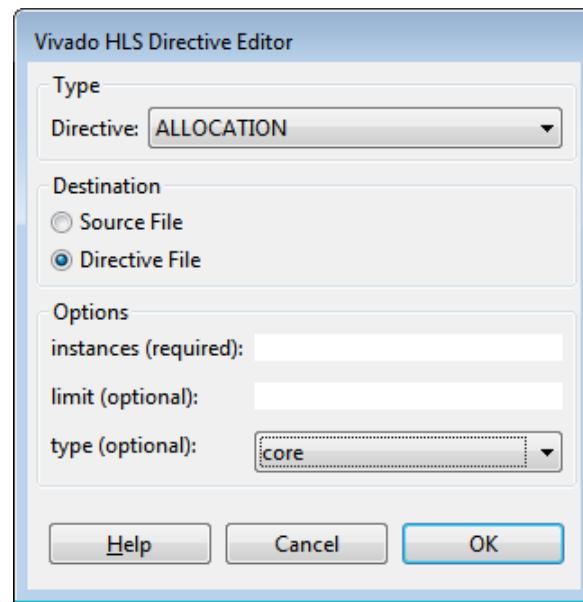
- Allocation directive
 - Can be used to limit the number of operations in scheduling and binding stages
- Resource directive
 - Can be used to specify which cores are using during binding
- Binding configuration
 - Can be used to minimize the number of operations

Slide 8-9:

Allocation: Limit the Numbers (1)



- Allocation directive limits different types of objects
 - Type: Operation
 - Instances are the operators
 - add, mul, urem, etc.
 - Type: Core
 - Instances are the cores
 - addsub, Mul2S, etc.
 - Operators and cores are listed in the *Vivado Design Suite High-Level Synthesis User Guide* (UG902)
 - Type: Functions
 - Instances of functions in the code
 - Discussed in more detail later

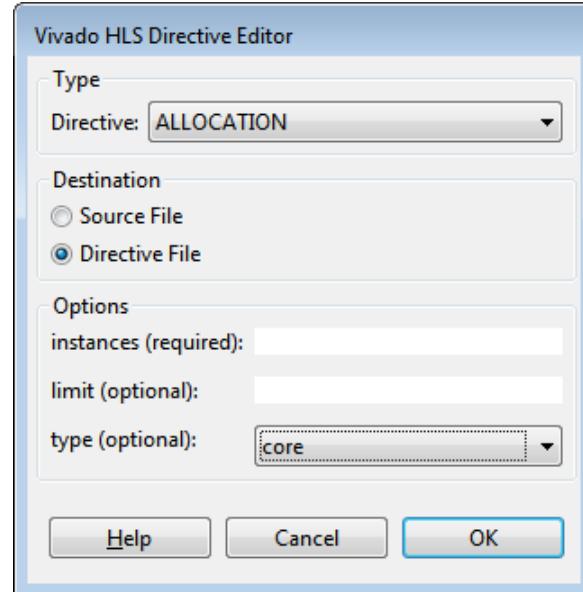


Slide 8-10:

Allocation: Limit the Numbers (2)



- Allocations are defined for a scope
 - Like all directives, allocations are set for the scope they are applied in
 - If the directive is applied to a function, loop, or region, it does not include objects outside of that function, loop, or region

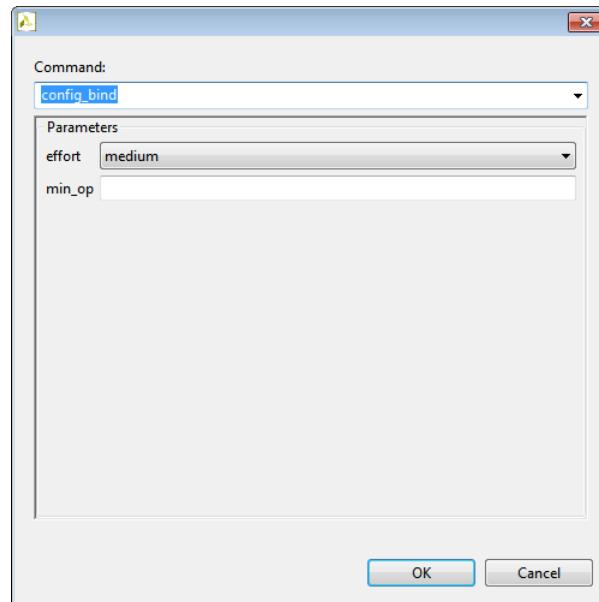


Slide 8-11:

Configuring Binding (1)



- Binding is controlled via a configuration command
 - Effort levels determine how much time is spent trying to map many operators onto fewer cores
 - As with all effort levels, they are worth using if you can see the design close to what is required
 - Else the tool will spend time exploring for possibilities and simply increase run time
- Use efforts judiciously
- Binding can be configured to minimize specific operators

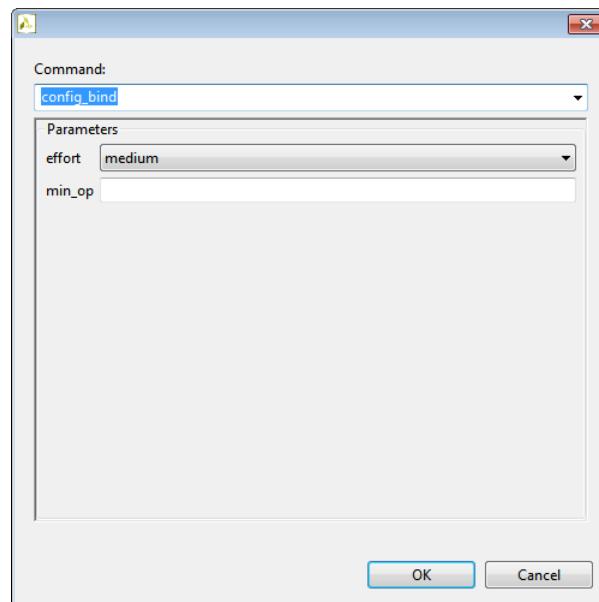


Slide 8-12:

Configuring Binding (2)



- Can be used to direct Vivado HLS tool to synthesize with the minimum number of operations
- Configuration command overrides multiplexing costs and can be used to force sharing
 - Works on all scopes in a design

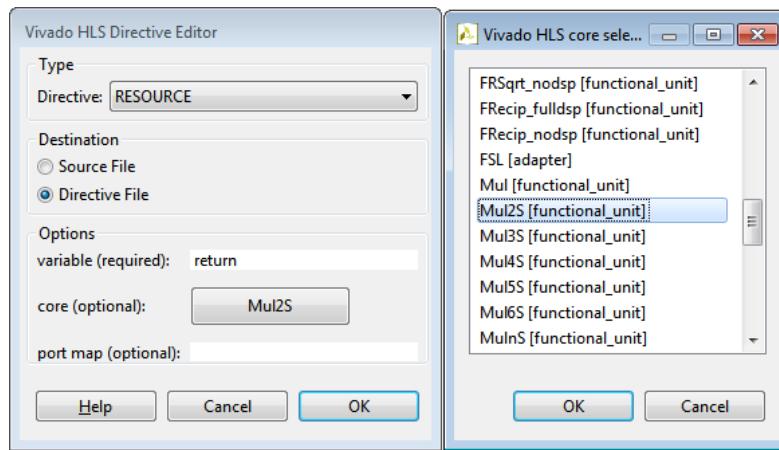


Slide 8-13:

Additional Control: Specify Resources

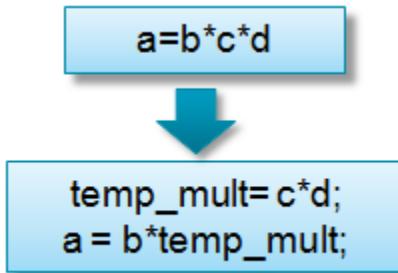


- User control of resources
 - Resource directive provides control over the specific resource (core) used to implement operations
 - Select the scope and right-click to apply the directive
 - Select **core** for a list of resources



In this example, variable temp_mult is implemented with a two-stage pipelined multiplier

- Specify the variable
- Caveat: multiple line coding
 - If multiple operations occur on a single line a temporary variable is required to isolate the specific operation



Slide 8-14:

Controlling the Structure of the Design



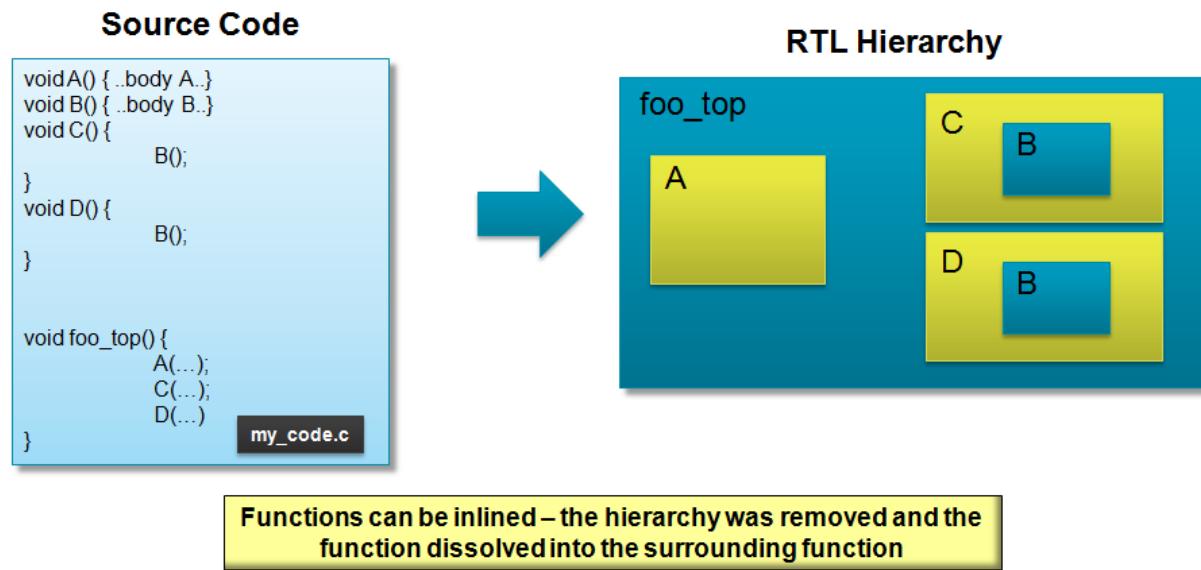
- Controlling the Resources Used
- **Controlling the Structure of the Design**
- Importance of Arbitrary Precision Types
- Summary

Slide 8-15:

Functions and RTL Hierarchy



- Each function is translated into an RTL block
 - Verilog module, VHDL entity

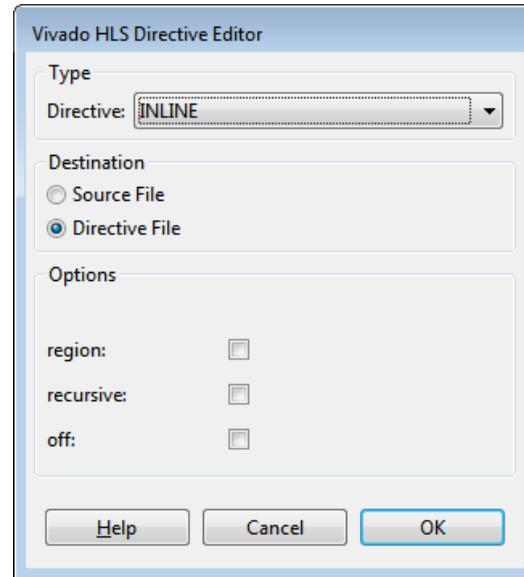


Slide 8-16:

Controlling Inlining (1)



- Vivado HLS tool performs some inlining automatically
 - This is performed on small logic functions if Vivado HLS tool determines area or performance will benefit
- User control
 - Functions can be specifically inlined
 - Function itself is inlined
 - Optionally recursively down the hierarchy
 - Optionally everything within a region can be inlined
 - Everything named as a region, function, or loop
 - Optionally inlining can be explicitly prevented

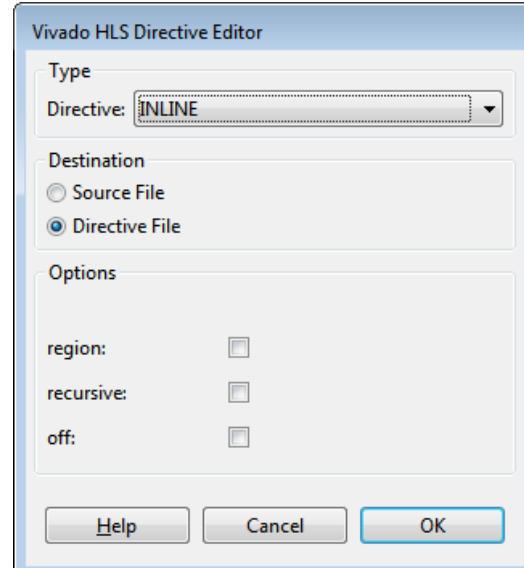


Slide 8-17:

Controlling Inlining (2)



- Turn inlining off
- Inlining functions allows for greater optimization
 - Like ungrouping RTL hierarchies: optimization across boundaries
 - Like ungrouping RTL hierarchies, it can result in lots operations and impact run time

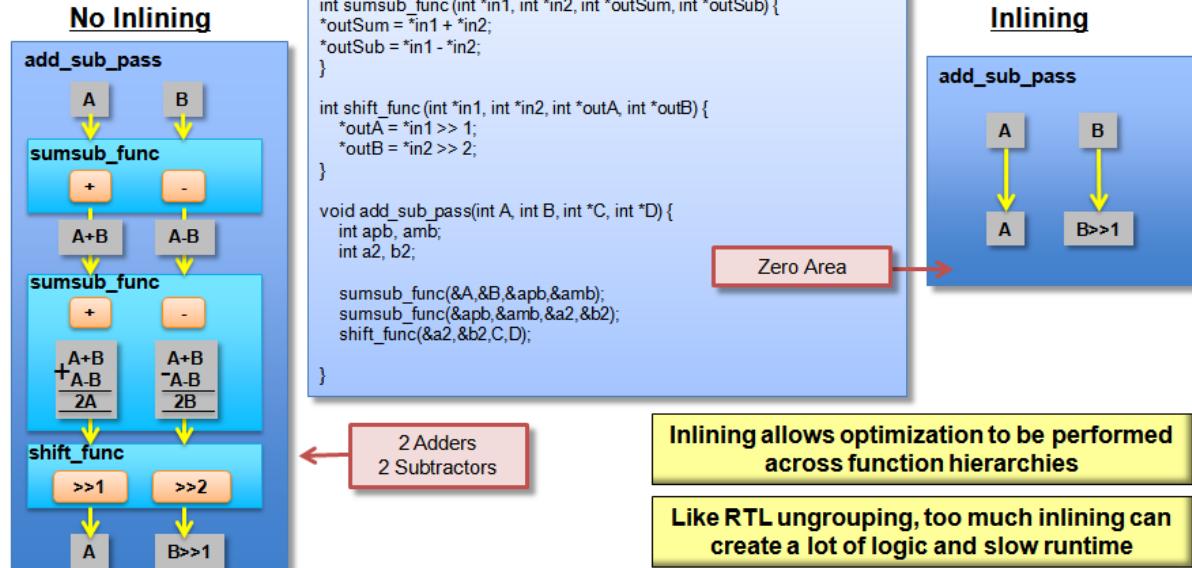


Slide 8-18:

Function Inlining



- Inlining can be used to remove function hierarchy



Slide 8-19:

Inline and Allocation: Shape the Hierarchy



Easy to Share

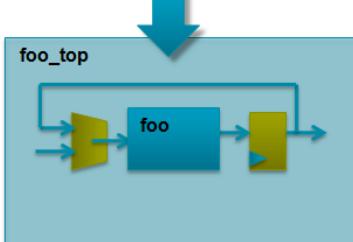
```

void foo() {
}

void foo_top() {
    foo(...);
    foo(...);
}

```

set_directive_allocation -limit 1
-type function foo_top foo



One RTL block is reused for both instances of function foo

Cannot be Shared

```

void dummy1() {
    foo();
}

void dummy2() {
    foo();
}

void foo_top() {
    dummy1(...);
    dummy2(...);
}

```

set_directive_allocation -limit 1
-type function foo_top foo

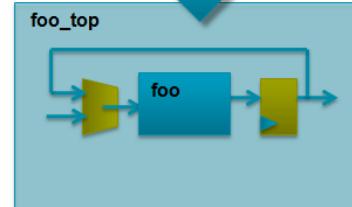


Function foo is not within the immediate scope of foo_top

Controlling Sharing

set_directive_allocation -limit 1
-type function foo_top foo

set_directive_inline dummy1
set_directive_inline dummy2



Inlining brings foo into function foo_top where it can be shared

Slide 8-20:

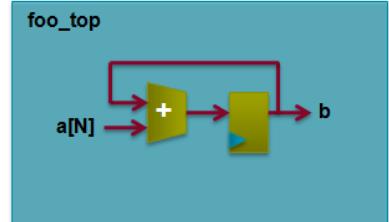
Loops



- By default, loops are rolled
 - Each C loop iteration → implemented in the same state
 - Each C loop iteration → implemented with same resources

```
void foo_top (...) {
...
Add: for (i=3;i>=0;i--) {
    b = a[i] + b;
}
...
```

Synthesis



- For area optimization
 - Keeping loops rolled maximizes sharing across loop iterations: each iteration of the loop uses the same hardware resources

Slide 8-21:

Loop Merging



- Loop merging can remove the redundant computation among multiple (related) loops
 - Improving area (and sometimes performance)

```
My_Region:{  
#pragma AP merge loop  
for (i = 0; i < N; ++i)  
    A[i] = B[i] + 1;  
for (i = 0; i < N; ++i)  
    C[i] = A[i] / 2;  
}
```

Merge

```
for(i = 0; i < N; ++i) {  
    A[i] = B[i] + 1;  
    C[i] = A[i] / 2;  
}
```

Effective code after compiler transformation

- Allows Vivado HLS tool to perform optimizations
 - Optimization cannot occur across loop boundaries

```
for (i = 0; i < N; ++i)  
    C[i] = (B[i] + 1) / 2;
```

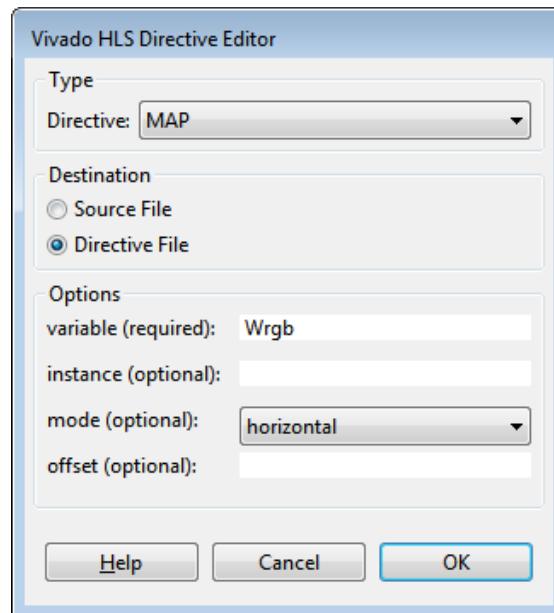
Removes A[i], any address logic, and any potential memory accesses

Slide 8-22:

Mapping Arrays (1)



- Arrays in the C model may not be ideal for the available RAMs
 - Code may have many small arrays
 - Array may not utilize the RAMs very well
- Array mapping
 - Mapping combines smaller arrays into larger arrays
 - Allows arrays to be reconfigured without code edits
 - Specify the array variable to be mapped
 - Give all arrays to be combined the same instance name

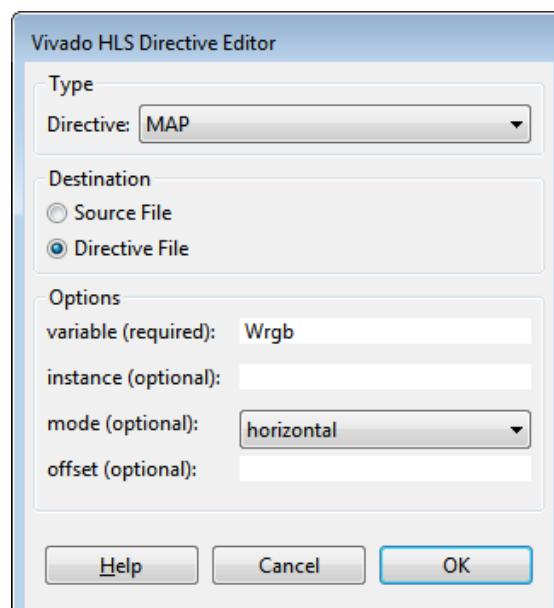


Slide 8-23:

Mapping Arrays (2)



- Vivado HLS tool provides options as to the type of mapping
 - Combine the arrays without impacting performance
 - Vertical and horizontal mapping
- Global arrays
 - When a global array is mapped, all arrays involved are promoted to global
 - When arrays are in different functions, the target becomes global
- Arrays which are function arguments
 - All must be part of the same function interface

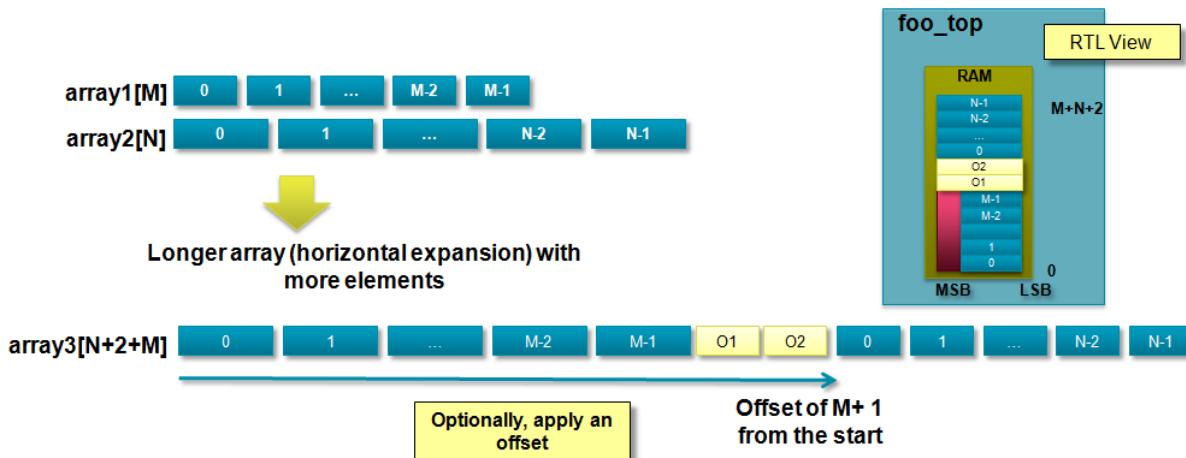


Slide 8-24:

Horizontal Mapping



- Horizontal mapping
 - Combines multiple arrays into longer (horizontal) array
 - Optionally allows the arrays to be offset
 - Default is to concatenate after the last element



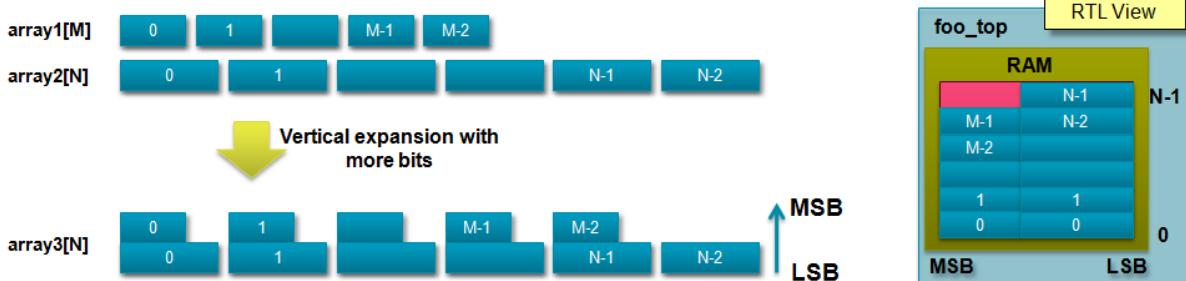
- First array specified (in GUI or Tcl script) starts at location zero

Slide 8-25:

Vertical Mapping



- Vertical mapping
 - Combines multiple arrays into an array with more bits



- First array specified (in Tcl or GUI) starts at the LSB

- Vertical mapping for performance
 - Creates RAMs with wide words → parallel accesses

Slide 8-26:

Importance of Arbitrary Precision Types

- Controlling the Resources Used
- Controlling the Structure of the Design
- ▪ **Importance of Arbitrary Precision Types**
- Summary

Slide 8-27:

Arbitrary Precision Integers



- 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.
 - Result in hardware that is not bit accurate and can give sub-standard QoR
- Vivado HLS tool provides bit-accurate types in both C and C++
 - SystemC types can be used in C++
 - Allow any arbitrary bit width to be specified and will simulate with bit accuracy

```
#include ap_cint.h
my_code.c
void foo_top (...) {
    int1          var1;      // 1-bit
    uint1         var1u;     // 1-bit
    unsigned
    int2          var2;      // 2-bit
    ...
    int1024       var1024;   // 1024-bit
    uint1024      var1024;   // 1024-bit unsigned
    ...
}
```

```
my_code.cpp
#include ap_int.h
my_code.cpp
void foo_top (...) {
    ap_int<1>           var1;      // 1-
    bit
    ap_uint<1>           var1u;     // 1-
    bit unsigned
    ap_int<2>           var2;      // 2-
    bit
    ...
    ap_int<1024>         var1024;   // 1024-bit
    ap_int<1024>         var1024u;  // 1024-bit unsigned
    ...
}
```

Slide 8-28:

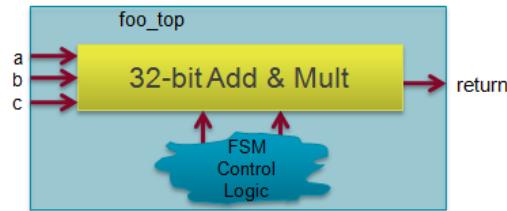
Why are Arbitrary Precision Types Needed?



- Code using native C int type

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

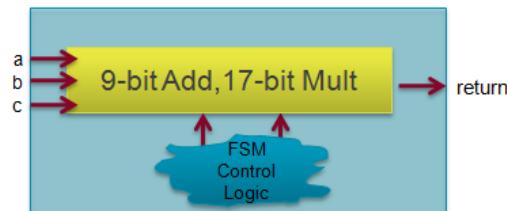
Synthesis →



- However, if the inputs will only have a max range of 8 bits
 - Arbitrary precision data types should be used

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

Synthesis →



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

Slide 8-29:

Summary



- Controlling the Resources Used
- Controlling the Structure of the Design
- Importance of Arbitrary Precision Types
- **Summary**

Slide 8-30:

Optimizing for Area



- Major area optimization techniques
 - Minimize bit widths
 - Map smaller arrays into larger arrays
 - Make better use of existing RAMs
 - Control loop hierarchy
 - Control function call hierarchy
 - Control the number of operators and cores

Slide 8-31:

Summary



- Controlling the resources used
 - Allocation and binding controls can define how many resources are used
- Controlling the structure of the design
 - Inlining functions: direct impact on RTL hierarchy and optimization possibilities
 - Loops: direct impact on reuse of resources
 - Arrays: direct impact on the RAM
- Importance of arbitrary precision types

Capture Your Notes Here



Introduction to the HLx Design Flow

Slide 9-1:

2016.1



This module describes the traditional RTL flow versus the Vivado® HLx design flow.
10 minutes.

Slide 9-2:

Objectives



After completing this module, you will be able to:

- Identify the opportunity to improve productivity by using a high-level language for a hardware design

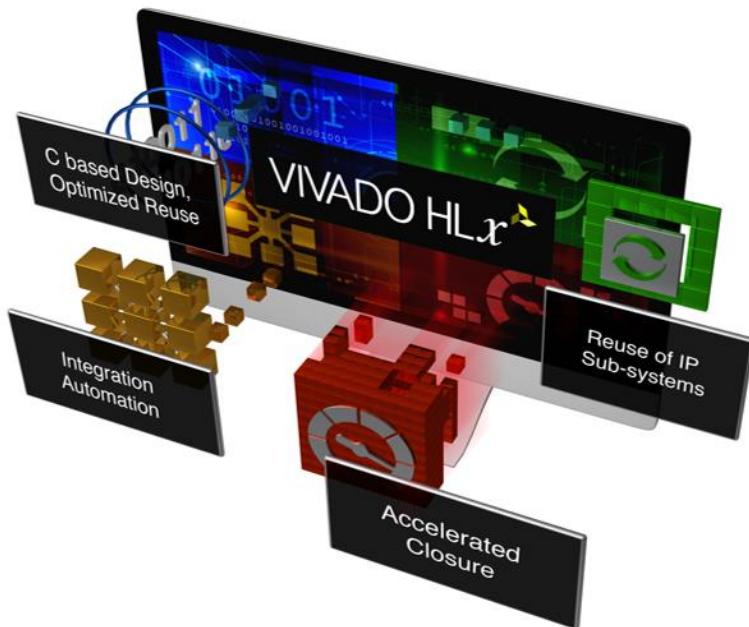
- Describe the differences between the traditional HDL design flow and a C-based design flow

Slide 9-3:

Productivity Improvement Opportunity

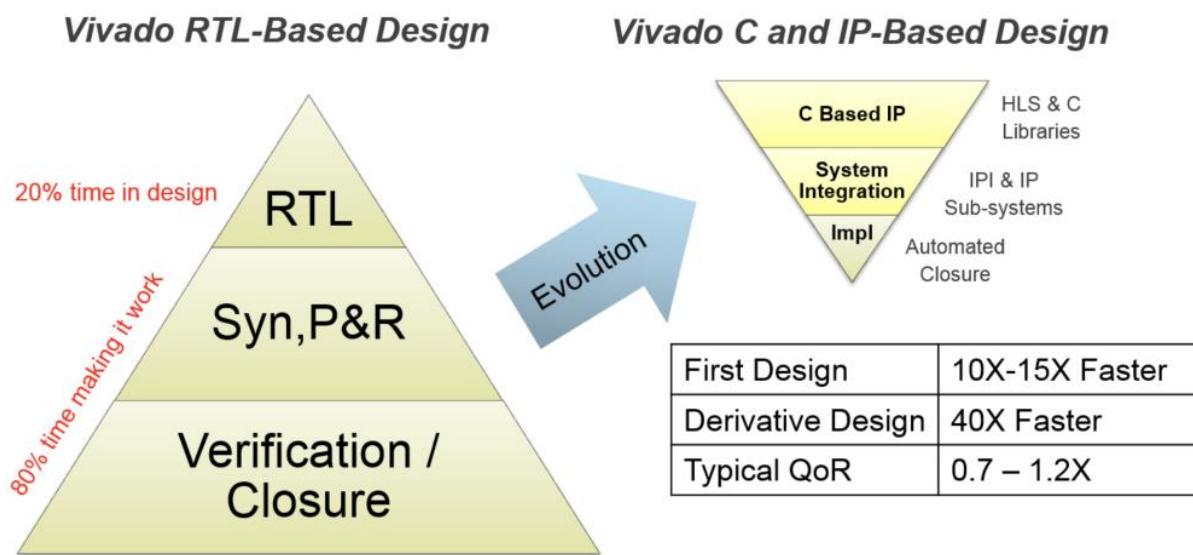


- Productivity gap with today's typical HDL flow
 - Requires hardware design expertise
 - Slow for design and simulation
 - Analogous to assembly code programming
- Leveraging unique Xilinx advantages: HLS and IPI
 - 15X productivity gain over HDL



Slide 9-4:

RTL vs C-Based Design



Traditional design development starts with experienced system architects estimating how their design will be implemented in a new technology and capturing both the system connectivity requirements and the value added differentiated logic in a high-level modeling format. In turn, RTL designs implement those requirements.

RTL design cycles typically consist of verification and design closure iterations for each block, as well as for the entire design. As a consequence of this methodology, the platform connectivity design never stabilizes, as any change in the differentiated logic can cause an I/O interface (e.g. DDR memory, Ethernet, or PCIe® core) to fail timing requirements. Also, RTL verification cycles no longer allow for exhaustive functional tests prior to hardware bring up.

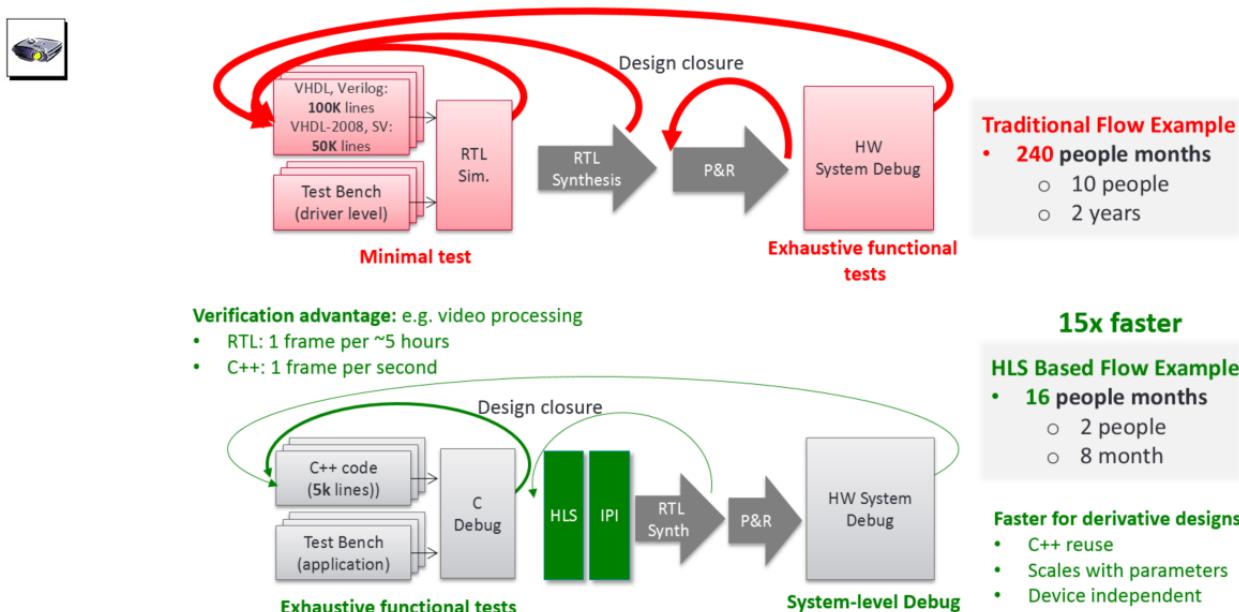
A high-level design methodology turns the development effort on its head, allowing designers to spend more time designing the value-add logic and less time trying to make it work. This flow provides a 15X reduction in design cycle compared to an RTL design flow.

The main attributes of a high-level methodology are:

- Separation of platform development and differentiated logic, allowing designers to focus on the company's high-value functionality.
- Rapid configuration, generation and closure of the platform connectivity, using the Vivado® IP Integrator with board awareness, as well as the Vivado IP systems.
- C-based simulation for the differentiated logic, decreasing simulation times by orders of magnitude over traditional RTL simulation.
- High-level synthesis with the Vivado HLS tool and C/C++ libraries, as well as the Vivado IP integrator for rapid implementation and system integration from C to silicon.

Slide 9-5:

Traditional RTL vs Vivado HLx Design Flow





A typical system starts with a software model of the system. Whether for entertainment, gaming, communications, or medicine, most products begin as a software model or prototype. This model is then distributed to the hardware and embedded software teams. Hardware design teams are tasked to choose an RTL microarchitecture that meets the system requirement.

The biggest advantage of programmable devices such as FPGAs is the ability to create custom hardware that is optimized for any specific application. As a result, the end product has orders of magnitude better performance per watt than a pure software program running on a distributed processor-based system.

The Vivado High-Level Synthesis (HLS) compiler provides a programming environment similar to those available for processor compilers. The main difference is that the Vivado HLS tool compiles the C code into an optimized RTL microarchitecture, while processor-based compilers generate assembly code to be executed on a fixed, GHz rate, processor architecture.

System architects, software programmers, or hardware engineers can use the Vivado HLS tool to create custom hardware optimized for throughput, power, and latency. This allows for optimized implementation of high performance, low power, or low cost systems for any application, including compute, storage, or networking.

The Vivado HLS tool accelerates design implementation and verification by enabling C/C++ specifications to be directly synthesized into VHDL or Verilog RTL after exploring a multitude of micro-architectures based on design requirements.

Functional simulation can be performed at that level, providing orders of magnitude acceleration over VHDL or Verilog simulation. For example, with a video motion estimation algorithm, the C input to the Vivado HLS tool executes 10 frames of video data in 10 seconds, while the corresponding RTL model takes roughly two days to process the same 10 video frames.

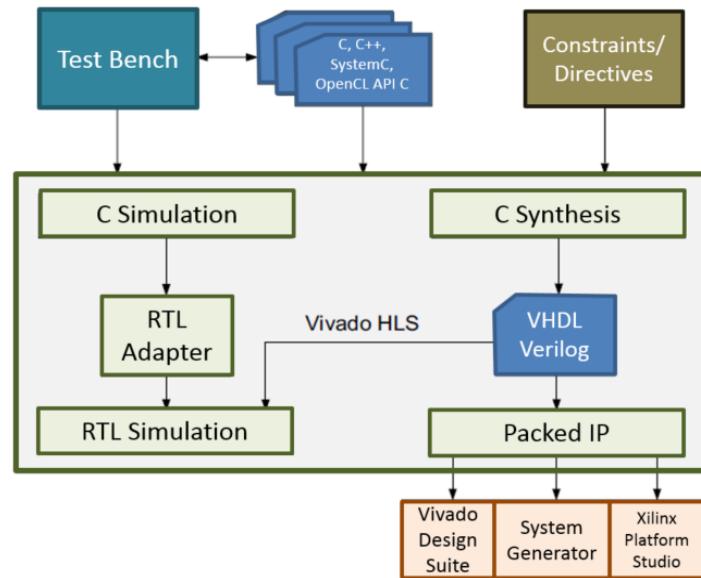
When coupled with the Vivado IP integrator, the Vivado HLS tool provides designers and system architects with a faster and more robust way of delivering quality designs.

Slide 9-6:

Vivado HLS Tool Flow



- Start with C, C++, SystemC, or OpenCL C
 - Design source and testbench code
- C to RTL synthesis
 - Schedule and map resources
 - Generate generic RTL code
 - Optimize using synthesis directives
- RTL validation
 - Uses the original C testbench
 - RTL simulation (XSim/ISim/ModelSim)
- Export to
 - IP catalog
 - System Generator
 - PCore



The following are the *inputs* to the Vivado HLS tool:

- **C function written in C, C++, SystemC, or an OpenCL API C kernel:** This is the primary input to the Vivado HLS tool. The function can contain a hierarchy of sub-functions.
- **Constraints:** Constraints are required and include the clock period, clock uncertainty, and FPGA target. The clock uncertainty defaults to 12.5% of the clock period if not specified.
- **Directives:** Directives are optional and direct the synthesis process to implement a specific behavior or optimization.
- **C testbench and any associated files:** The Vivado HLS tool uses the C testbench to simulate the C function prior to synthesis and to verify the RTL output using C/RTL co-simulation.

The following are the *outputs* from the Vivado HLS tool:

■ **RTL implementation files in hardware description language (HDL) formats:**

This is the primary output from the Vivado HLS tool. Using Vivado synthesis, you can synthesize the RTL into a gate-level implementation and an FPGA bitstream file. The RTL is available in the following-industry standard formats:

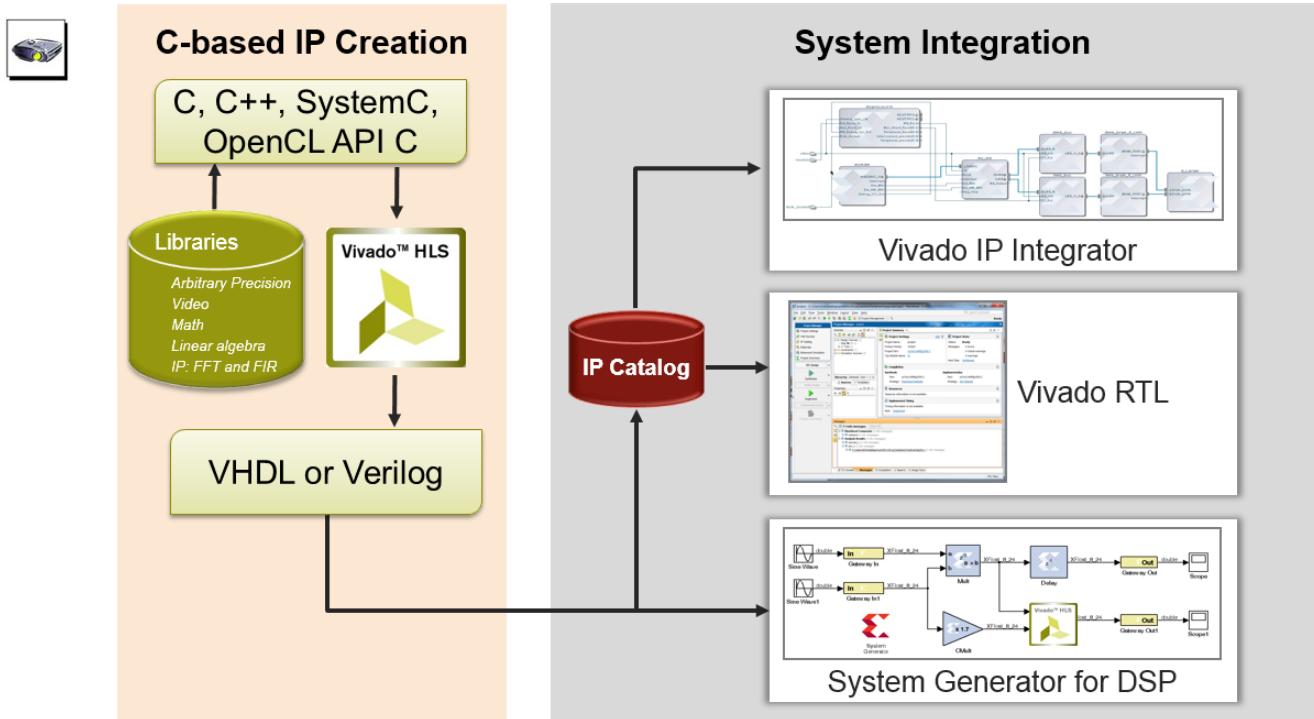
- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)

The Vivado HLS tool packages the implementation files as an IP block for use with other tools in the Xilinx design flow. Using logic synthesis, you can synthesize the packaged IP into an FPGA bitstream.

■ **Report files:** This output is the result of synthesis, C/RTL co-simulation, and IP packaging.

Slide 9-7:

User-Preferred System Integration Environment

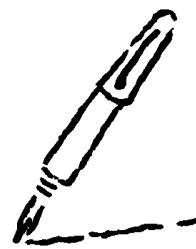


Here is a more in-depth, or system view, of the Vivado HLS tool. As you can see, the input to the Vivado HLS tool is C, C++, SystemC, or OpenCL API C.

In this diagram, you will also notice that the Vivado HLS C and C++ are including C Libraries and a *math.h*, some video libraries, or some DSP libraries available as part of the Vivado HLS tool library. The Vivado HLS tool uses a number of C libraries to improve the performance of the designs.

When the output of the Vivado HLS tool is available, it can be exported into the IP catalog and throughout the rest of the Vivado Design Suite. It can be imported as an RTL block into Vivado RTL, used in the Vivado IP integrator, and be used as a single block in a system generator for DSP.

Capture Your Notes Here



HLS vs. SDSoc Development Environment Flow

Slide 10-1:

2016.1



This module describes the HLS flow versus the SDSoc™ development environment flow. 15 minutes.

Slide 10-2:

Objectives



After completing this module, you will be able to:

- Identify the opportunity to improve productivity by using a high-level language for a hardware design

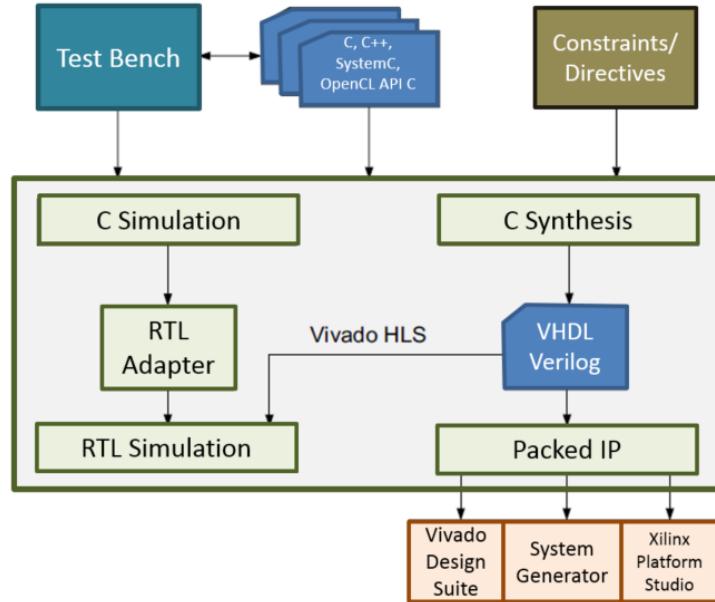
- Describe the differences between the HLS design flow and SDSoc development environment flow

Slide 10-3:

Vivado HLS Tool Flow



- Start with C, C++, SystemC, or OpenCL C
 - Design source and testbench code
- C to RTL synthesis
 - Schedule and map resources
 - Generate generic RTL code
 - Optimize using synthesis directives
- RTL validation
 - Uses the original C testbench
 - RTL simulation (XSim/ISim/ModelSim)
- Export to
 - IP catalog
 - System Generator
 - PCore



The following are the *inputs* to the Vivado HLS tool:

- **C function written in C, C++, SystemC, or an OpenCL API C kernel:** This is the primary input to the Vivado HLS tool. The function can contain a hierarchy of sub-functions.
- **Constraints:** Constraints are required and include the clock period, clock uncertainty, and FPGA target. The clock uncertainty defaults to 12.5% of the clock period if not specified.
- **Directives:** Directives are optional and direct the synthesis process to implement a specific behavior or optimization.
- **C testbench and any associated files:** The Vivado HLS tool uses the C testbench to simulate the C function prior to synthesis and to verify the RTL output using C/RTL co-simulation.

The following are the *outputs* from the Vivado HLS tool:

■ **RTL implementation files in hardware description language (HDL) formats:**

This is the primary output from the Vivado HLS tool. Using Vivado synthesis, you can synthesize the RTL into a gate-level implementation and an FPGA bitstream file. The RTL is available in the following-industry standard formats:

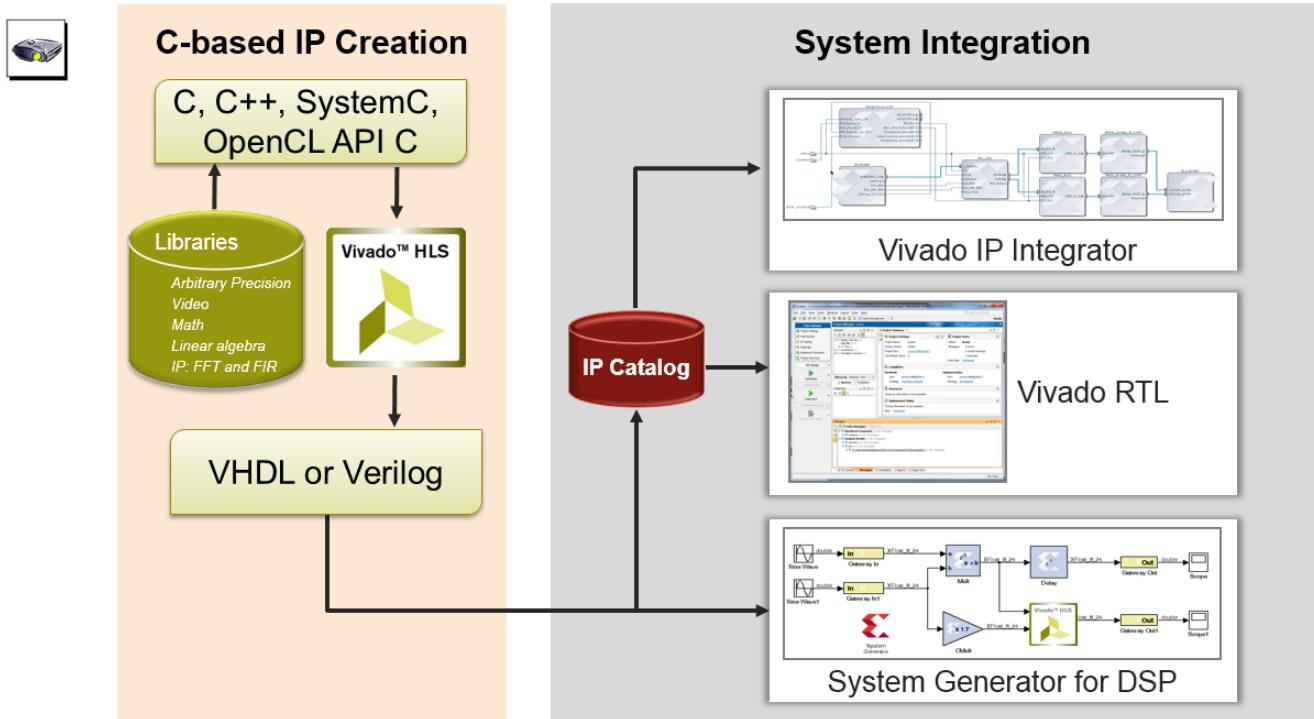
- VHDL (IEEE 1076-2000)
- Verilog (IEEE 1364-2001)

The Vivado HLS tool packages the implementation files as an IP block for use with other tools in the Xilinx design flow. Using logic synthesis, you can synthesize the packaged IP into an FPGA bitstream.

■ **Report files:** This output is the result of synthesis, C/RTL co-simulation, and IP packaging.

Slide 10-4:

User-Preferred System Integration Environment



Here is a more in-depth, or system view, of the Vivado HLS tool. As you can see, the input to the Vivado HLS tool is C, C++, SystemC, or OpenCL API C.

In this diagram, you will also notice that the Vivado HLS C and C++ are including C Libraries and a *math.h*, some video libraries, or some DSP libraries available as part of the Vivado HLS tool library. The Vivado HLS tool uses a number of C libraries to improve the performance of the designs.

When the output of the Vivado HLS tool is available, it can be exported into the IP catalog and throughout the rest of the Vivado Design Suite. It can be imported as an RTL block into Vivado RTL, used in the Vivado IP integrator, and be used as a single block in a system generator for DSP.

Slide 10-5:

Scope of the SDSoc Development Environment



- What is it?
 - The SDSoc tool is designed to quickly identify and target C/C++ software functions to programmable logic
 - Implements software functions into hardware as accelerators
 - Abstracts the controls of other tools (Vivado® HLS tool, IPI, SDK and the Vivado Design Suite)
 - The SDSoc tool builds the entire infrastructure: Processing system, accelerator cores, device drivers, data movers, signaling, etc.
 - By contrast, the Vivado HLS tool only builds the RTL – connectivity, supporting IP, drivers, etc. must be supplied by the designer



The industry continues to demand more performance with less power. Often processors struggle with numerically intense algorithms and designers are often forced to contend with lower performance or more computational resources to manage this problem. SoCs offer the best of both worlds: powerful processors as well as programmable logic.

Functions that burden processors can be offloaded to the programmable logic by converting them into "hardware functions", or accelerators. These accelerators can be pipelined or placed in parallel for even greater performance.

Slide 10-6:

Benefits of Using the SDSoc Development Environment



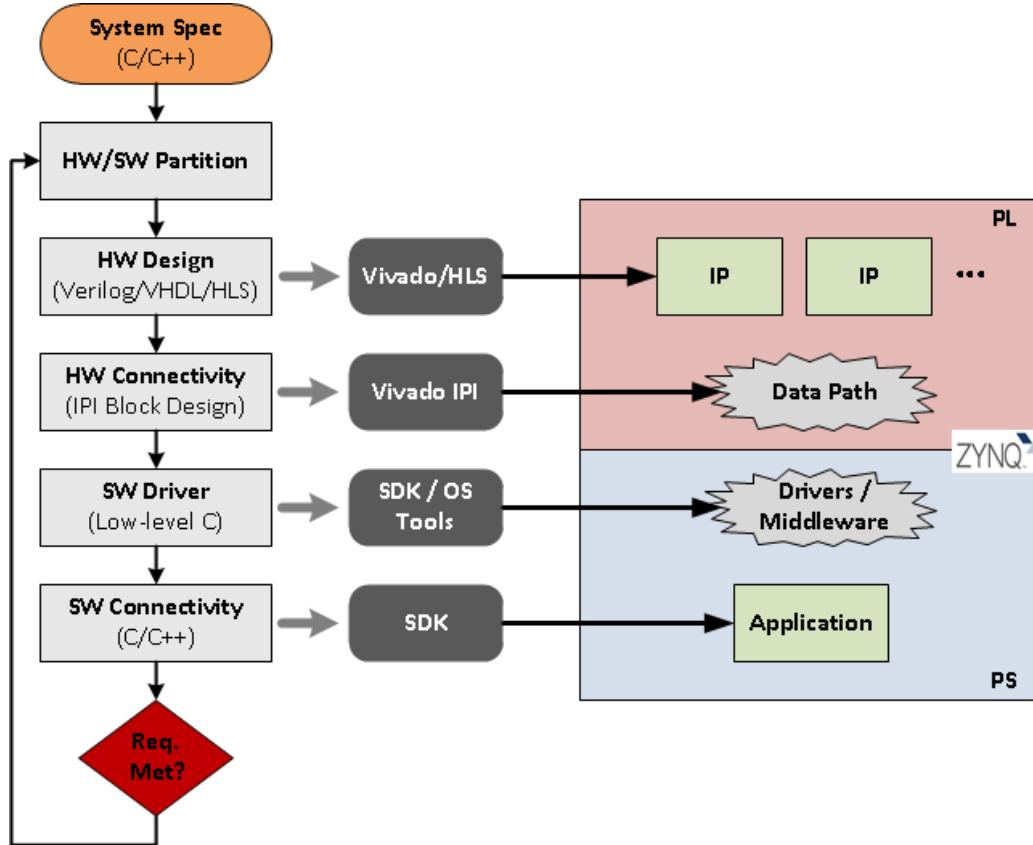
- Shorter development cycles
 - Estimation shows improvement over software-only solution for system and accelerators
 - Iterative improvement can be made at early stages of development without the need to build hardware
- Simplified interface and partitioning between hardware and software
 - Software-based flow vs RTL – user interfaces with software-only tools; hardware management abstracted
 - Removes much of the hardware/software distinction and throw-over-the-wall effect
- Automated initial design
 - Users can tweak code at macro- and micro-architectural levels
 - Designers still have manual control over the constructed Vivado HLS tool and Vivado Design Suite projects

Slide 10-7:

Development Flow Without the SDSoc Tool



- SoCs reduce some challenges thanks to tight integration between the hardware and software
 - Overall process requires expertise at all steps in the process



Targeting an SoC requires a number of tools and demands that developers have significant experience with each.

The designer is responsible for every aspect of the system including:

- Isolating bottlenecks (i.e., critical points) in software performance.
- Designing the hardware that will be used to accelerate software.
- Determining communication mechanisms between software and hardware components.
- Minimizing the (often) large investment associated with migrating software to hardware.

For example:

- IP can be created using a combination of Vivado HLS tool projects, System Generator, and RTL-based Vivado Design Suite projects.
- Connectivity between the PS and PL is accomplished in a Vivado IP integrator block design, and architecture is completely manual.
- Drivers can be created using third-party tools supported by the OS.
- The top-level application can be managed in SDK.

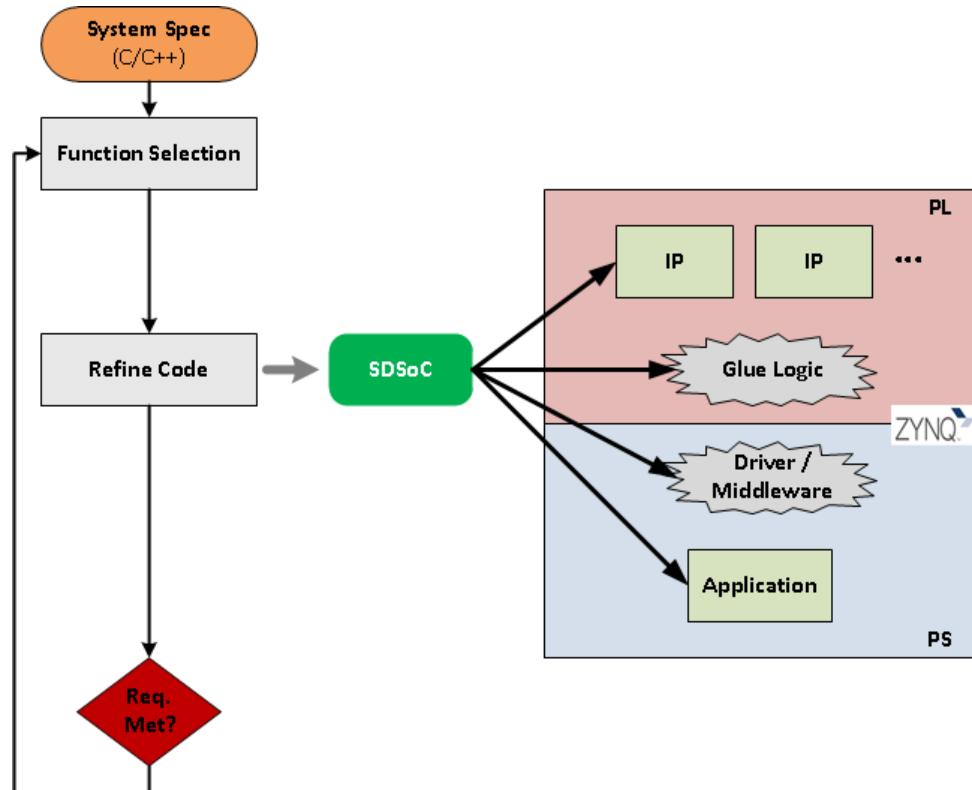
Bottom line: Users must be expert in every step and with every tool to have a reasonable chance of building an effective system.

Slide 10-8:

Development Flow Using the SDSoC Tool



- SDSoC development environment consolidates a multi-step/multi-tool process into a single tool and reduces hardware/software partitioning to simple function selection
 - Code typically needs to be refined to achieve optimal results





Note that the same tools that were previously involved in this flow (e.g., Vivado HLS tool, Vivado IPI, SDK) are still included. Now, however, the SDSoc development environment manages them in the background rather than requiring the user to interface with them directly. The SDSoc tool automatically selects data paths and builds drivers as part of the flow without the need for user interaction—thus reducing required experience and development time.

Designers can focus on the function of the system rather than the mechanics.

The SDSoc development environment simplifies the design of hardware by supplying the hardware:

- Only user-selected functions are compiled to a hardware accelerator.
- Accelerator cores are either pulled from a library or generated on the fly by calling the Vivado HLS tool in the background.
- If a prepackaged hardware implementation of a software function is not found, then the Vivado HLS tool is used to create the core.
- Prepackaged cores can come from various sources (e.g., supplied with a base platform or individually from third parties).

There is also no need to create or manage hardware projects:

- Separately maintained Vivado Design Suite projects are no longer required.
- No RTL (i.e., Verilog/VHDL) coding is required.
- Overall development flow remains in the C/C++ domain.

C-callable IP allows direct mapping of existing hardware IP that users could still build using RTL if they prefer. There are designers who would still want the flexibility to craft their accelerator and also benefit from the SDSoc development environment flow.

Although refining the code is not strictly necessary, it is typically required for optimal results. Such code refinements may or may not require changes to the original code structure. For example, a less intrusive change may involve adding some pragmas while a more intrusive change could involve unrolling a loop.

Key aspects of the SDSoc development environment that greatly simplify the development flow include:

- Transforming C/C++ programs into complete hardware/software systems targeting a Zynq® All Programmable SoC.
- Building the entire infrastructure including DMAs (data movers), device drivers, and bitstream.
- Automatically calling required hardware design tools for hardware generation.

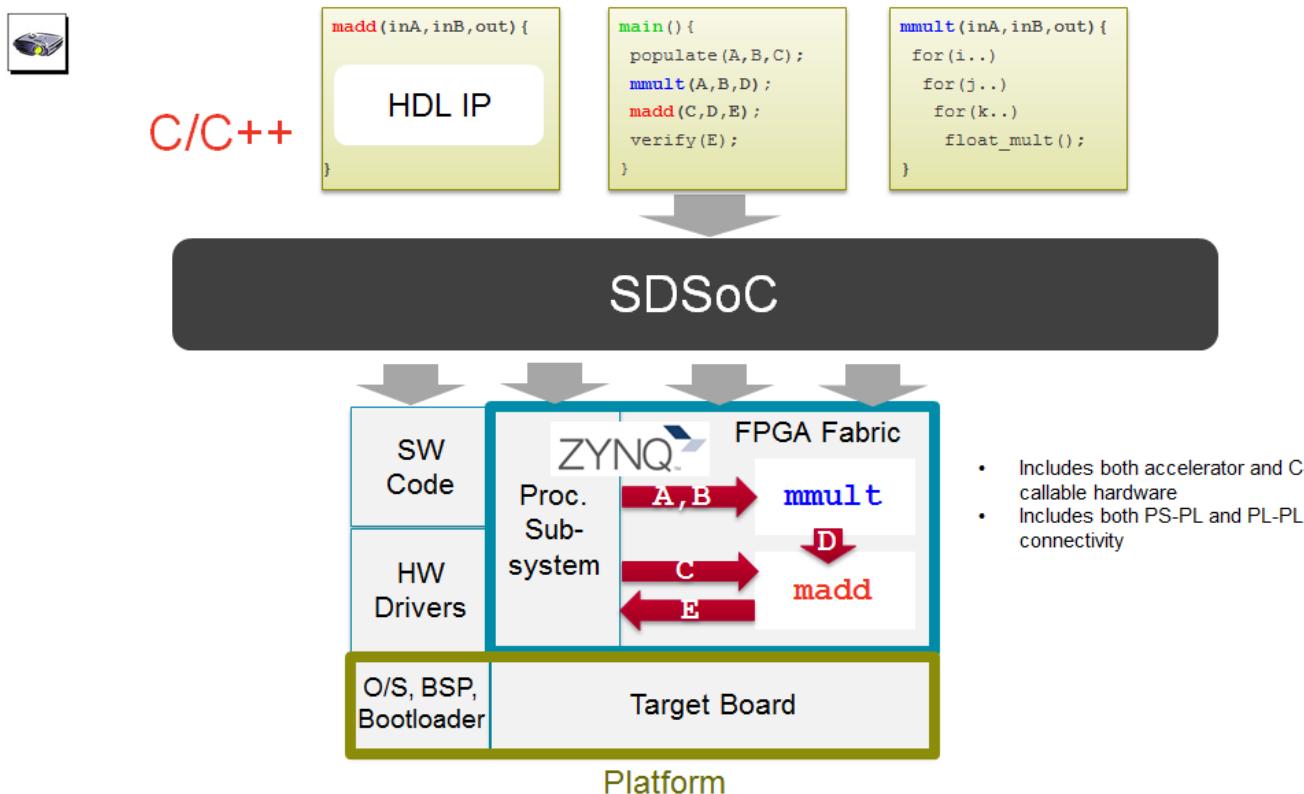
- Bonus: SDSoC development environment can enable user RTL to be C-callable from a Zynq AP SoC.
 - Leverages existing or third-party IP.

The SDSoC development environment delivers increased performance in an accelerated development cycle:

- 10:1 software/hardware speed-up ratios are common.
- Elimination of hardware design from development flow yields faster results (e.g., week versus month).
- Multiple iterations of system-level changes can be done quickly and efficiently.

Slide 10-9:

Process and Results of a Typical SDSoC Development Environment Process



The emphasis on this slide is that software goes into the magical black box and a mixed hardware/software design pops out the back end.

This example illustrates the case of two matrix math functions selected for hardware acceleration.

Starting from the top:

- 1) C/C++ project has the functions **mmult** and **madd** that are selected for hardware acceleration. In the case of **madd**, a pre-defined accelerator core is already available.
- 2) The project is built using the SDSoc development environment, which analyzes the code and generates a complete hardware/software system. In the case of **mmult**, which only had a software definition, an accelerator core is generated.
- 3) At the bottom of the diagram is an illustration of the generated system, which consists of application code and drivers in the processing subsystem, as well as data-flow/control interfaces (A,B,C,D,E) and accelerators in the programmable logic subsystem. Interfaces A and C represent control, while B, D, and E represent the data path. Notice how the SDSoc development environment allows data to flow between accelerators rather than treating them independently. The fact that SDSoc generates systems based on a platform is also highlighted in this illustration where a platform would define, among other things, a targeted board and supporting software.

The SDSoc development environment extends pre-defined platforms by adding the necessary hardware (accelerators and data path logic) and software (drivers and software stubs) to migrate the selected functions to hardware.

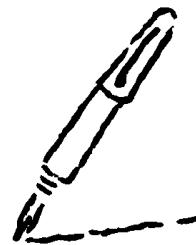
Slide 10-10:

What Users Need to Know to Be Highly Successful



- Tools abstract away many of the challenges
 - Novices can attain some basic level of results
 - Full performance only comes with understanding
- Vivado HLS tool
 - SDSoc development environment does not perform any code optimizations
 - Vivado HLS tool skills are very important to achieve optimal performance
 - Understanding the Zynq AP SoC architecture is also beneficial
- C/C++
 - Tools are GNU based
 - New C learners should follow a non-SDSoC development environment flow
 - Familiarity with refactoring for the SDSoc development environment

Capture Your Notes Here



Vivado HLS Tool: C Code

Slide 11-1:

2016.1

This module describes the Vivado® HLS tool support for the C languages, as well as arbitrary precision data types. It also describes working with pointers in the design and hardware modeling with streaming data types. 75 minutes.

Slide 11-2:

Objectives



After completing this module, you will be able to:

- Describe the support offered by the Vivado HLS tool for C, C++, and SystemC
- Write an efficient C-type testbench
- List the unsupported constructs and workarounds to implement a C-type testbench
- Define bit-accurate operators using arbitrary precision types
- Describe the modeling issues when using pointers
- Generate efficient hardware from C sources using stream and shift register classes

Slide 11-3:

C Validation Flow



- **C Validation Flow**
 - C Language Support
 - Data Types and Bit Accuracy
 - Pointers
 - Hardware Modeling
 - OpenCV Libraries
 - Summary

Slide 11-4:

Comprehensive C Support



- A complete C validation and verification environment
 - Vivado HLS tool supports complete bit-accurate validation of the C model
 - Vivado HLS tool provides a productive C-RTL co-simulation verification solution
- Vivado HLS tool supports C, C++, SystemC, and OpenCL API C Kernel
 - Functions can be written in any version of C
 - Wide support for coding constructs in all three variants of C
 - Easier to discuss what is not supported than what is



Starting with Vivado Design Suite 2015.1, the Vivado HLS tool supports OpenCL API C language constructs and built-in functions from the OpenCL API C 1.0 embedded profile.

The Vivado HLS tool supports the following standards for C compilation/simulation:

- ANSI-C (GCC 4.6)
- C++ (G++ 4.6)
- OpenCL API (1.0 embedded profile)
- SystemC (IEEE 1666-2006, version 2.2)

Slide 11-5:

Library Support (1)



- Arbitrary Precision Data Types library
 - Support for both integer (*ap_cint.h*, *ap_int.h*) and fixed-point (*ap_fixed.h*) arbitrary precision data types
- HLS MATH library (*hls_math.h*)
 - Extensive support for the synthesis of the standard C (*math.h*) and C++ (*cmath.h*) libraries
 - Support includes floating-point and fixed-point functions
- HLS Stream library (*hls_stream.h*)
 - Provides a C++ template class `hls::stream<>` for modeling streaming data structures
- HLS Video library (*hls_video.h*)
 - OpenCV video function support
 - Libraries target real-time, full HD video processing



You can use each of the C libraries in your design by including the library header file. These header files are located in the include directory in the Vivado HLS tool installation area.

Note: The header files for the Vivado HLS tool C libraries do not have to be in the include path if the design is used in the Vivado HLS tool. The paths to the library header files are automatically added.

Slide 11-6:

Library Support (2)



- HLS IP library
 - Instantiates and parameterizes the FIR compiler (*hls_fir.h*) and FFT LogiCORE™ IP (*hls_fft.h*) as function calls from your C++ code
 - Implements a shift register using a Xilinx SRL primitive (*ap_shift_reg.h*)
 - Implements a Xilinx direct digital synthesizer (DDS) from your a C++ code (*hls_dds.h*)
- HLS Linear Algebra library (*hls_linear_algebra.h*)
 - Provides a number of commonly used linear algebra functions
- HLS DSP library



HLS IP libraries:

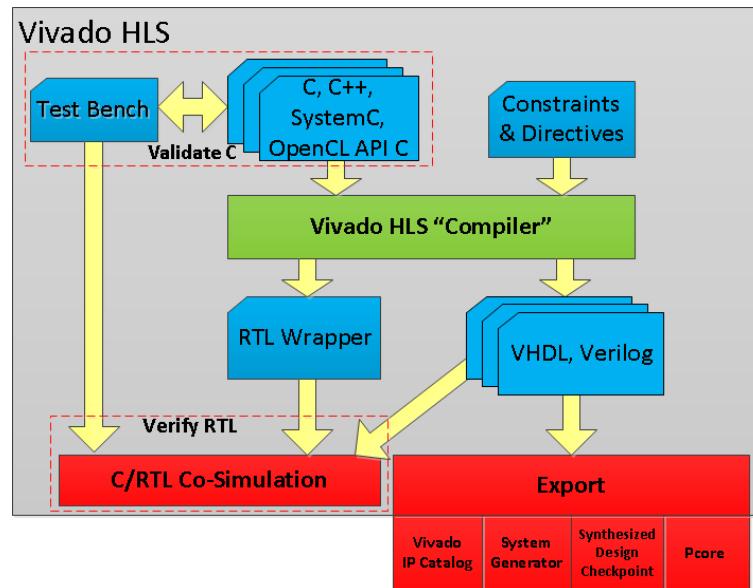
- *hls_fft.h*: Allows the Xilinx LogiCORE IP FFT to be simulated in C and implemented using the Xilinx LogiCORE block.
- *hls_fir.h*: Allows the Xilinx LogiCORE IP FIR to be simulated in C and implemented using the Xilinx LogiCORE block.
- *ap_shift_reg.h*: Provides a C++ class to implement a shift register that is implemented directly using a Xilinx SRL primitive.

Slide 11-7:

C Validation and RTL Verification



- Two steps to verifying the design
 - Pre-synthesis: C **validation**
 - Validate the algorithm is correct
 - Post-synthesis: RTL **verification**
 - Verify the RTL is correct



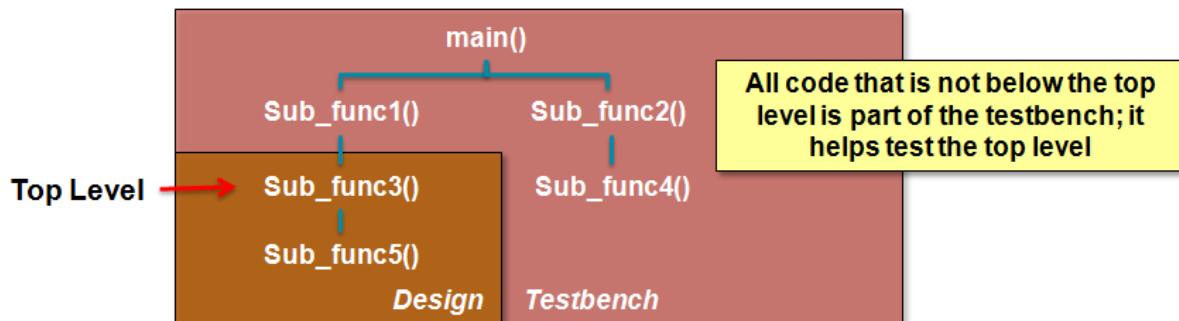
- C validation
 - A HUGE advantage to using HLS
 - Fast, free verification
 - Validating the algorithm is correct **before** synthesis
 - Follow the given testbench tips
- RTL verification
 - Vivado HLS tool can co-simulate the RTL with the original testbench

Slide 11-8:

Testbenches



- "Testbench" is any code above the top level
 - Top of any C function is function main()
 - Function main() cannot be the top level for synthesis



- Project files
 - Ideally the testbench and the design are in different files
 - Add the top level as a design file: `add_file my_top.c`
 - Add the testbench as a testbench file: `add_file -tb my_testbench.c`
 - If they are in the same file, add the same file as both a design file and testbench file

Slide 11-9:

C Function Testbench



- Testbench is the level above the function
 - The *main()* function is **above** the function to be synthesized
- Good practices
 - Testbench should compare the results with golden data
 - Automatically confirms any changes to the C are validated
 - Automatically verifies the RTL is correct
 - Testbench should return a 0 if the self-checking is correct
 - Anything but a 0 (zero) will cause RTL verification to issue a FAIL message
 - Function *main()* should expect an integer return (non void)

```
int main () {  
    int ret=0;  
    ...  
    ret = system("diff --brief -w output.dat output.golden.dat");  
  
    if (ret != 0) {  
        printf("Test failed !!!\n");  
        ret=1;  
    } else {  
        printf("Test passed !\n");  
    }  
    ...  
    return ret;  
}
```

Slide 11-10:

Testbench: Gotcha 1



- Use a RETURN value
 - RTL verification re-uses the C testbench to verify the RTL design

```

@I[LIC-101] Checked in features[AUTOESL_FLOWAUTOESL_OPT AUTOESL_SC AUTOESL_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.
03
12
21
30
@E[SIM-3] Simulation failed: test bench return error code "20".
@E[SIM-1] *** AutoSim finished: FAIL ***

```

The testbench *could* show the correct results after RTL sim

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

- If the testbench does not return a 0, RTL sim will say it failed
- Examine the SIM-3 message
 - Message SIM-3 says why the RTL failed verification
 - Return of "20" means nothing returned: likely a bad testbench

```

@E[SIM-3] Simulation failed: test bench return error code "20".
@E[SIM-1] *** AutoSim finished: FAIL ***

```

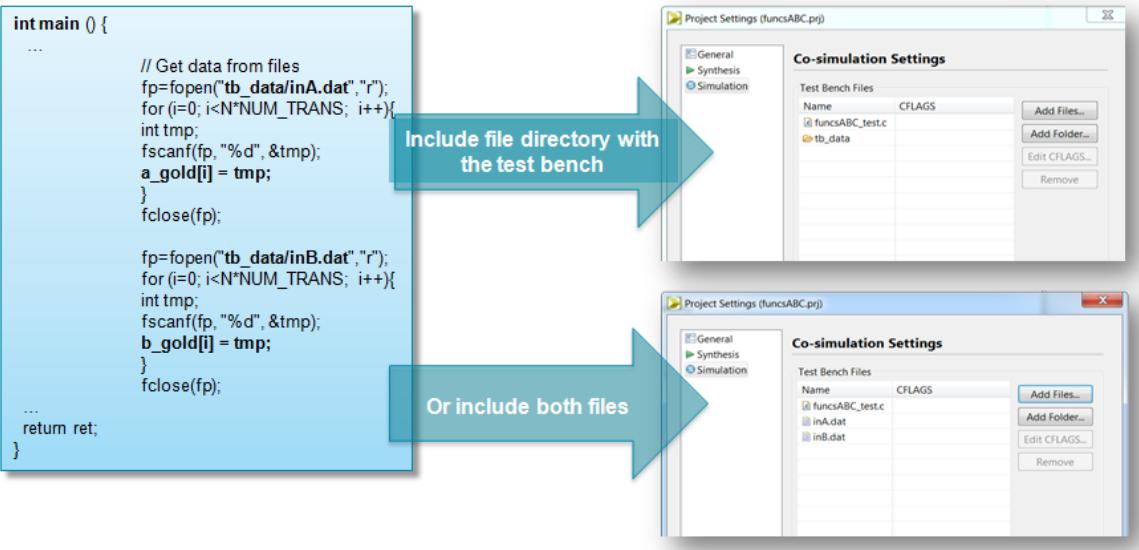
- Return of "1" shows self checking was used: valid failure
 - If the testbench returns "1" when self checking fails (as in previous example)

Slide 11-11:

Testbench: Gotcha 2



- Include all files used by the testbench in the Vivado HLS tool
 - This testbench reads golden data from files



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

Slide 11-12:

Determine or Create the Top-Level Function



- Determine the top-level function for synthesis
- If there are multiple functions, they must be merged
 - There can only be one top-level function for synthesis

Given a case where functions func_A and func_B are to be implemented in FPGA

Re-partition the design to create a new single top-level function inside main()

main.c

```
int main () {
    ...
    func_A(a,b,*i1);
    func_B(c,*i1,*i2);
    func_C(*i2,ret)
    ...
    return ret;
}
```

main.c

```
#include func_AB.h
int main (a,b,c,d) {
    ...
    // func_A(a,b,i1);
    // func_B(c,i1,i2);
    func_AB(a,b,c, *i1, *i2);
    func_C(*i2,ret)
    ...
    return ret;
}
```

Recommendation is to separate testbench and design files*

*Else add file as design file **and** testbench

func_AB.c

```
#include func_AB.h
func_AB(a,b,c, *i1, *i2) {
    ...
    func_A(a,b,*i1);
    func_B(c,*i1,*i2);
    ...
}
```

Slide 11-13:

Understand the I/O



■ At a glance

```
int foo_top(a, b, c, *i1, ar[2], *i2) {  
    ...  
    func_A(a,b,*i1);  
    func_B(c,*i1,*i2);  
    func_C(*i2,ret)  
  
    return ret;  
}
```

Throughout this module, everything
that says pointer also replies to a
reference in C++

- Pass-by-value arguments will only ever be input ports
 - There must be a pointer, array, or function return to get an output port
- Any function return will result in output port ap_return at the RTL
 - Only guaranteed valid when ap_done is high on the last cycle of the function
- Pointers that are read become inputs
- Pointers that are written to become outputs
 - Pointers that are read and written to become both (two ports)
- Arrays will become RAM or FIFO ports
 - Or can be partitioned into individual element ports

Slide 11-14:

C Language Support



- C Validation Flow
- **C Language Support**
- Data Types and Bit Accuracy
- Pointers
- Hardware Modeling
- OpenCV Libraries
- Summary

Slide 11-15:

C, C++, and SystemC Support



- Vast majority of C, C++, and SystemC is supported
 - Provided it is statically defined at **compile time**
 - If not defined until **run time**, it will not be synthesizable
- Any of the three variants of C can be used
 - If C is used, Vivado HLS tool 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
 - Single output: function return

Slide 11-16:

FIR Filter with C (1)



- C coding style
 - Top-level function defines the I/O ports
 - If required, sub-functions can be added

```
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_coeff.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.c

fir_coeff.h

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

Slide 11-17:

FIR Filter with C (2)



- Storage
 - Static: required for anything that must hold its value between invocations
 - 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 that are 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_coeff.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,

```

Slide 11-18:

Arrays to RAMs: Initialization (1)



- 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

Slide 11-19:

Arrays to RAMs: Initialization (2)



- 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
- Array can be initialized as a static

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



- A coefficient array does not need to be initialized every time but doing that costs "nothing" in software, unlike hardware
- Statics are initialized at "start up" only
 - In C, in the RTL, and via bitstream

Slide 11-20:

FIR Filter with C++



- C++ coding style
 - Top-level defines RTL I/O
 - 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

```
#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

```
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;
```

CFir functionality defined

```
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;
```

Top-level function defines the I/O class CFIR is instantiated as fir1

```
return fir1(x);
}
```

Slide 11-21:

FIR Filter with SystemC



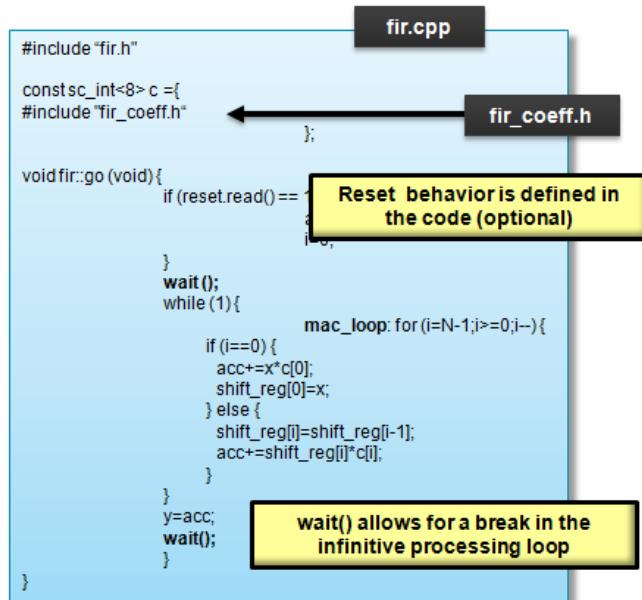
- SystemC style
 - Header defines the I/O ports
 - Constructor calls the functions
 - Can be SC_CTHREAD sensitive on the clock (with optional reset)
 - Can be SC_METHOD sensitive on any signal
 - Multiple threads, methods, and functions can be used

```
SC_MODULE(fir){
//-----Input Ports-----
sc_in <bool>
sc_in <bool>
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);
}
};
```

I/O defined

**Clock-sensitive thread (SC_CTHREAD)
SC_METHOD is also supported (combo)
SC_THREAD is not supported**



Slide 11-22:

Unsupported Constructs: Overview



- System calls and function pointers
 - Dynamic memory allocation
 - *malloc()*, *alloc()*, and *free()*
 - Standard I/O 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



Slide 11-23:

Unsupported: Dynamic Memory Allocation (1)



- Dynamic memory allocation
 - Not allowed because it requires the construction (or destruction) of hardware at runtime
 - *malloc*, *alloc*, *free* are **not** 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];
```

Slide 11-24:

Unsupported: Dynamic Memory Allocation (2)



- 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);
```

Slide 11-25:

Unsupported: System Calls



- System calls
 - Most C system calls do not have hardware counterparts and are not synthesizable
 - printf(), getc(), time(), ...
 - Vivado HLS tool 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 Vivado HLS tool 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  
    ...  
}
```

Slide 11-26:

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 *is allowed* 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;  
}
```

Slide 11-27:

Unsupported: Recursive Functions



- Avoid recursive functions
 - Not synthesizable in general
 - 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

Slide 11-28:

Data Types and Bit Accuracy



- C Validation Flow
- C Language Support
- **Data Types and Bit Accuracy**
 - C apint Types
 - C++ ap_int Types
 - C++ ap_fixed Types
 - SystemC Types
 - Floating-Point Types
- Pointers
- Hardware Modeling
- OpenCV Libraries
- Summary

Slide 11-29:

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.
 - Result in hardware that is not bit accurate and can give sub-standard QoR
- Vivado HLS tool 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 that there is no loss of accuracy

Slide 11-30:

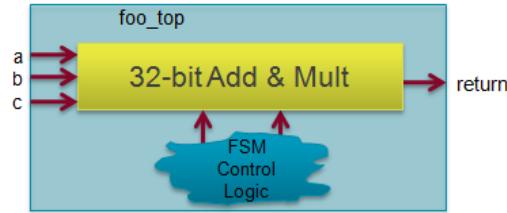
Why are Arbitrary Precision Types Needed?

- Code using native C int type



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

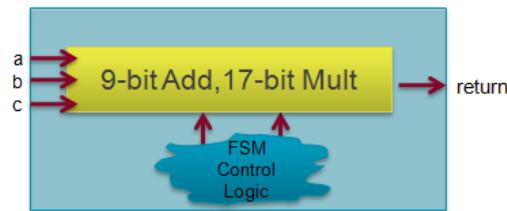
Synthesis



- However, if the inputs will only have a max range of 8 bits
 - Arbitrary precision data types should be used

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

Synthesis



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

Slide 11-31:

HLS and C Types



- Four basic types that can be used for HLS
 - Standard C/C++ types
 - Vivado HLS tool enhancements to C: ap_cint
 - Vivado HLS tool 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)

Slide 11-32:

C apint Types

- C Validation Flow
- C Language Support
- **Data Types and Bit Accuracy**
 - **C apint Types**
 - C++ ap_int Types
 - C++ ap_fixed Types
 - SystemC Types
 - Floating-Point Types
- Pointers
- Hardware Modeling
- OpenCV Libraries
- Summary



Slide 11-33:

Arbitrary Precision: C apint Types



- For C
 - Vivado HLS tool 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

Slide 11-34:

C apint Types: Bit Selection and Manipulation



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

Slide 11-35:

C++ ap_int Types

- C Validation Flow
- C Language Support
- **Data Types and Bit Accuracy**
 - C apint Types
 - **C++ ap_int Types**
 - C++ ap_fixed Types
 - SystemC Types
 - Floating-Point Types
- Pointers
- Hardware Modeling
- OpenCV Libraries
- Summary



Slide 11-36:

Arbitrary Precision: C++ ap_int Types



- For C++
 - Vivado HLS tool types ap_int can be used
 - Range: 1 to 1024 bits
 - Signed: ap_int<W>
 - Unsigned: ap_uint<W>
 - Bit width is specified by W
- At the command line
 - Use g++ at the Vivado HLS tool CLI (shell)
 - Include the path to the Vivado HLS tool header file
 - shell> g++ -o my_test test.c test_tb.c -I\$VIVADO_HLS_HOME/include
 - makefile is included in the labs

```
#include ap_int.h  
  
void foo_top (...) {  
    ap_int<9> var1;           // 9-bit  
    ap_uint<10> var2;         // 10-bit  
    unsigned
```

Include header file

Slide 11-37:

Microsoft Visual Studio Support (1)



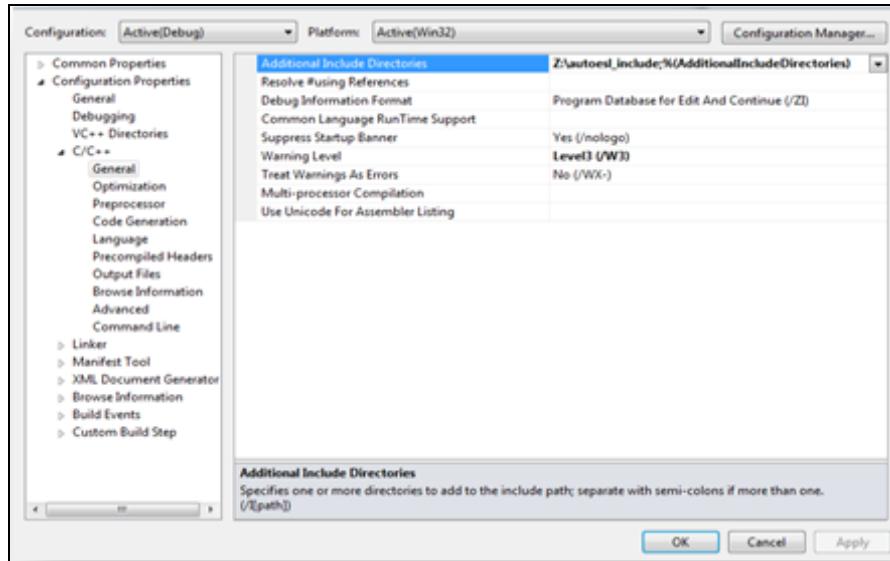
- C++ arbitrary precision types are supported in the Microsoft Visual Studio Compiler
 - Simply include the Vivado HLS tool directory \$(Vivado_HLS_HOME)/include
 - **Note:** C designs using arbitrary precision types (apint) must still use apcc
- C++ designs using AP_INT types
 - In the Microsoft Visual Studio project

Slide 11-38:

Microsoft Visual Studio Support (2)



- 1) Click **Project**
- 2) Click **Properties**
- 3) In the panel that shows up, select **C/C++**
- 4) Select **General**
- 5) Click additional include directories and add the path



Slide 11-39:

AP_INT Operators and Conversions



- Fully supported for all arithmetic operators

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

- Methods for type conversion

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

Slide 11-40:

AP_INT 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]
(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();

Slide 11-41:

C++ ap_fixed Types

- C Validation Flow
- C Language Support
- **Data Types and Bit Accuracy**
 - C apint Types
 - C++ ap_int Types
 - **C++ ap_fixed Types**
 - SystemC Types
 - Floating-Point Types
- Pointers
- Hardware Modeling
- OpenCV Libraries
- Summary



Slide 11-42:

Arbitrary Precision: C++ ap_fixed Types



- Support for fixed-point data types in C++
 - Include the path to the *ap_fixed.h* header file
 - Both signed (*ap_fixed*) and unsigned types (*ap_ufixed*)

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

\$AUTOESL_HOME/include/ap_fixed.h

- Advantages of fixed-point types
 - Result of variables with different sizes is automatically taken care of
 - Decimal point is automatically aligned
 - Quantization: Underflow is automatically handled
 - Overflow: Saturation is automatically handled

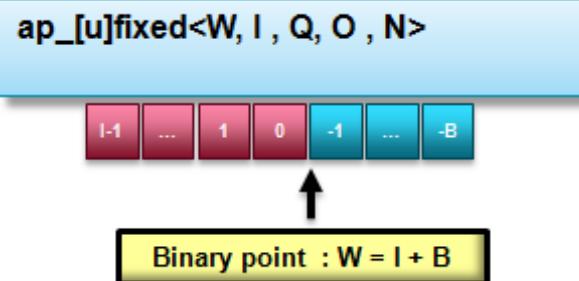
Alternatively, make the result variable large enough such that overflow or underflow does not occur

Slide 11-43:

Definition of ap_fixed Type



- Fixed-point types are specified by
 - Total bit width (W)
 - Number of integer bits (I)
 - Quantization/rounding mode (Q)
 - Overflow/saturation mode (O)
 - Number of saturation bits



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

Slide 11-44:

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_INF
 - AP_RND_ZERO, AP_RND_CONV
- Quantization modes (truncation)
 - AP_TRN, AP_TRN_ZERO

Slide 11-45:

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
 - 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
 - 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)

Slide 11-46:

Quantization Modes: Rounding to Infinity



- AP_RND: rounding to plus infinity
 - Rounds by adding the MSB of the removed bits to the remaining bits
 - Effect is to round towards plus infinity (increases in value)
 - 01.01 (1.25 using 4 bits) rounds to 01.1 (1.5 using 3 bits)
- AP_RND_MIN_INF: rounding to minus infinity
 - Always decreases the value of the number; just remove the redundant bits (truncate)
 - 01.01(1.25) → 01.0(1)
 - 10.11(-1.25) → 10.1(-1.5)
- AP_RND_INF: rounding to infinity
 - Positive numbers are rounded to plus infinity. MSB of deleted bits is added to the remaining bits
 - Negative numbers are rounded to minus infinity. Redundant bits truncated
 - 01.01(1.25) → 01.1(1.5) round up for positive numbers
 - 10.11(-1.25) → 10.1(-1.5) truncation for negative numbers

Slide 11-47:

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)

Slide 11-48:

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
 - Number of saturation bits; N is considered when wrapping

Slide 11-49:

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 two's complement notation, one more negative value than positive value can be represented
 - If the absolute values of MIN and MAX symmetrical around zero are needed, 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)

Slide 11-50:

Overflow Mode: Wrap



- AP_WRAP: wrap around overflow operation
 - Two types
 - With zero saturation bits (N parameter)
 - With non-zero saturation bits
- AP_WRAP, N = 0
 - Default overflow mode
 - Any MSB bits outside the range are deleted
 - For unsigned numbers, acts like a counter; after MAX wraps around to zero
 - For signed numbers, after MAX, the next value is MIN
 - 0100(4) → 100(-4)
 - 1011(-5) → 011(3)
- AP_WRAP, N > 0
 - When N > 0, N MSB bits are to be saturated or set to 1
 - Sign bit is retained, so positive numbers remain positive and negative numbers remain negative
 - Bits that are not saturated are copied starting from the LSB side

Slide 11-51:

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: Delete redundant MSBs – 0100(4) → 100(-4)
 - Step 2: 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

Slide 11-52:

AP_FIXED Operators and Conversions



- Fully supported for all arithmetic operators

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

- Methods for type conversion

Methods	Example
To integer	Convert to a integer type
To unsigned integer	Convert to an unsigned integer type
To 64-bit integer	Convert to a 64-bit long long type
To 64-bit unsigned integer	Convert to an unsigned long long type
To double	Convert to double type
To ap_int	Convert to an ap_int

Slide 11-53:

AP_FIXED Methods



- Methods for bit manipulation

Methods	Example
Length	Returns the length of the variable
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]

Slide 11-54:

SystemC Types

- C Validation Flow
- C Language Support
- **Data Types and Bit Accuracy**
 - C apint Types
 - C++ ap_int Types
 - C++ ap_fixed Types
 - **SystemC Types**
 - Floating-Point Types
- Pointers
- Hardware Modeling
- OpenCV Libraries
- Summary



Slide 11-55:

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.
- At the command line
 - Compile with g++
 - Include the SystemC files from the Vivado HLS tool tree*
- SC types
 - Can be used in C++ designs without the need to convert the entire design to SystemC
- SystemC support
 - Vivado HLS tool supports SystemC 1.3 Synthesizable subset*

* www.systemc.org/downloads/drafts_review

```
* 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
```

Slide 11-56:

Floating-Point Types

- C Validation Flow
 - C Language Support
 - **Data Types and Bit Accuracy**
 - C apint Types
 - C++ ap_int Types
 - C++ ap_fixed Types
 - SystemC Types
 - **Floating-Point Types**
 - Pointers
 - Hardware Modeling
 - OpenCV Libraries
 - Summary
- 

Slide 11-57:

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 tool supports the floating-point (FP) cores for each Xilinx technology
 - If Xilinx has an FP core, Vivado HLS tool supports it
 - It will automatically be synthesized
 - If there is no such FP core in the Xilinx technology
 - Design should be re-coded to fixed-point types
 - Certain math functions are supported (discussed later)

Slide 11-58:

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	-	-	-

Slide 11-59:

Using Floating-Point Types Example (1)



- 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++
- \$AUTOESL_HOME/include/ap_fixed.h

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

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

Required if it's a C++ function

Slide 11-60:

Using Floating-Point Types Example (2)

- 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
                                     // 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
int                  var_i = sqrtf(foo_i);   // Fixed → 32-bit sqrt core → float to fixed conversion
                                         // 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

Slide 11-61:

Support for Math Functions (1)



- Vivado HLS tool 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
- Vivado HLS tool 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
- Replace *math.h* or *cmath.h* with Vivado HLS tool function *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*)

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

Slide 11-62:

Support for Math Functions (2)



- 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

For more information on floating-point, refer to Application Note: "Floating-Point Design with Vivado HLS"

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

Slide 11-63:

Arbitrary Precision Flow



- Perform C validation
 - Run the testbench
 - Save the results to a golden file
- Add arbitrary precision types to the design
- C validation should be re-run to verify the results are identical
 - Debug any inconsistencies
- C (not C++ or SystemC) designs must use APCC for arbitrary precision validation
 - gcc compiler does not support arbitrary precision integers
 - APCC utility is supplied with the Vivado HLS tool
 - Command line compatible with gcc
 - Allows accurate bit-accurate simulation for C designs

Slide 11-64:

Pointers



- C Validation Flow
- C Language Support
- Data Types and Bit Accuracy
- **Pointers**
- Hardware Modeling
- OpenCV Libraries
- Summary

Slide 11-65:

Using Pointers



- 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
 - These issues were addressed in the "I/O Interfaces" module, but re-iterated here

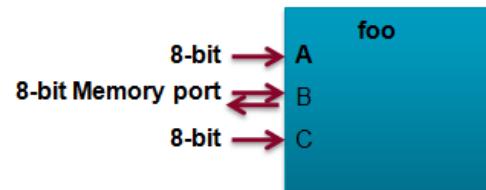
Slide 11-66:

Pointer and Structs



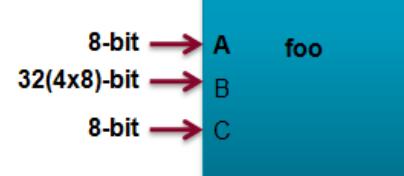
- Struct passed as a pointer
 - When a struct is passed as a pointer
 - Data will be accessed as pass-by-reference

```
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 as a value
 - 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);
```



Slide 11-67:

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;
}
```

Slide 11-68:

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;
    }
}
```

Value of “d” defined in the testbench

Pointer arithmetic

```
#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 and writes must be sequential (start from location 0 implied)
 - This functionality can only be implemented by using ap_bus
- Address generated by pointer arithmetic
 - Can only be exported externally by using ap_bus
 - All other pointer interfaces are streaming type

Slide 11-69:

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[]
- 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;
```

The diagram illustrates the flow of the workaround code. It shows four yellow boxes with arrows pointing to specific parts of the C code:

- Code for non-synthesis flow (simulation)**: Points to the `#ifndef __SYNTHESIS__` block.
- Fixed-size resource**: Points to the declarations `my_type _yuv[N];` and `my_type _scale[N];`.
- Point to new fixed-sized resource**: Points to the assignments `my_type *yuv = &_yuv;` and `my_type *scale = &_scale;`.
- Keep pointer in code**: Points to the final assignment `res = *yuv;`.

Slide 11-70:

Multi-Access Pointers: REQUIRED Volatile

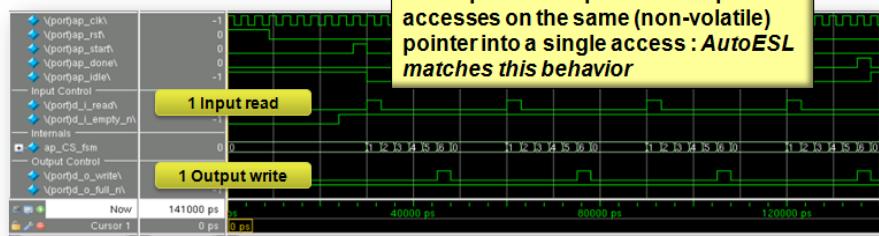
- Easier to see with a simple example



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

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

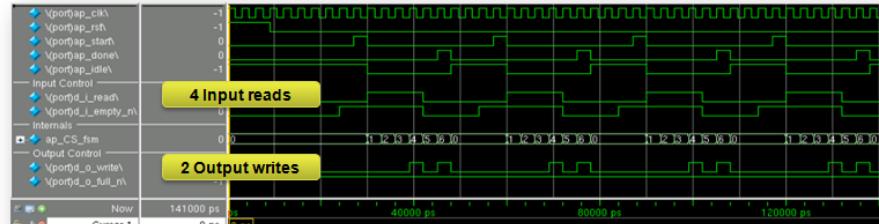
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```



```
voidfifo (
    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;
}
```



Slide 11-71:

Multi-Access Pointers: Limited Visibility



- Intermediate results are hard to use for verification
 - Only the final pointer value is passed to the testbench
 - Final value of *d_o can be captured in the testbench and compared
 - Intermediate values can only be seen by using printf: they cannot be automatically verified by the testbench

```
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;
        }
    }
}
```

- 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 can be used instead

Slide 11-72:

Multi-Access Pointers: Testbench 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;
    }
}
```

C testbench may correctly provide four values

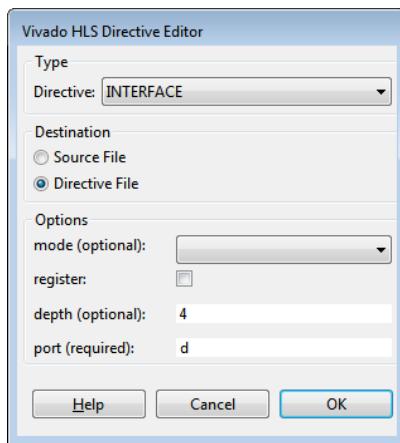
```
int main () {
    int d[5];
    int i;

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

    foo(d);

    return 0;
}
```

- AutoSim (RTL verification) only sees a pointer on the interface
 - And will create a SystemC co-simulation wrapper to supply or store one sample
 - Use the **depth** option to specify the actual depth
 - Vivado HLS tool 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 mismatch

Slide 11-73:

Hardware Modeling

- C Validation Flow
- C Language Support
- Data Types and Bit Accuracy
- Pointers
-  ■ **Hardware Modeling**
- OpenCV Libraries
- Summary

Slide 11-74:

Hardware Modeling



- In general, Vivado HLS tool is used to synthesize standard C
 - Written for software purposes
 - Written using semantics meant for software
 - Generally this is not an issue
 - ... but when performance is required
- Coding for performance
 - Regardless of the target you always re-code for performance
 - C code is optimized for when the target is DSP or real-time OS
 - Vivado HLS tool has wide support for C, C++, SystemC, and OpenCL API C
 - Code changes are typically to remove implied sub-optimal behavior in the code
- Vivado HLS tool supports coding constructs that better model and implement hardware
 - Streams
 - SRL shift registers

Slide 11-75:

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

Slide 11-76:

Using Streams



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

Slide 11-77:

Stream Example



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]`

- 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-blocking 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();
```

Slide 11-78:

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
 - Not just a shift register which *should* be inferred by the ISE® tool
 - **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

Slide 11-79:

Using ap_shift_reg (1)

- 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(ln1,2);
...
```

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)

Slide 11-80:

Using ap_shift_reg (2)

- 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, ln1;
bool En;

// Read location 3 of Sreg into var1
// THEN if En=1
// Shift all values up one and load ln1 into location 0
var1 = Sreg.shift(ln1,3,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, ln1;
bool En;

Sreg.shift(ln1);
```

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

Slide 11-81:

OpenCV Libraries

- C Validation Flow
 - C Language Support
 - Data Types and Bit Accuracy
 - Pointers
 - Hardware Modeling
 - **OpenCV Libraries**
 - Summary
- 

Slide 11-82:

Vivado HLS Tool Video Libraries (1)



- 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

Slide 11-83:

Vivado HLS Tool Video Libraries (2)



- 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

Slide 11-84:

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

Slide 11-85:

Video Library Supported Functions

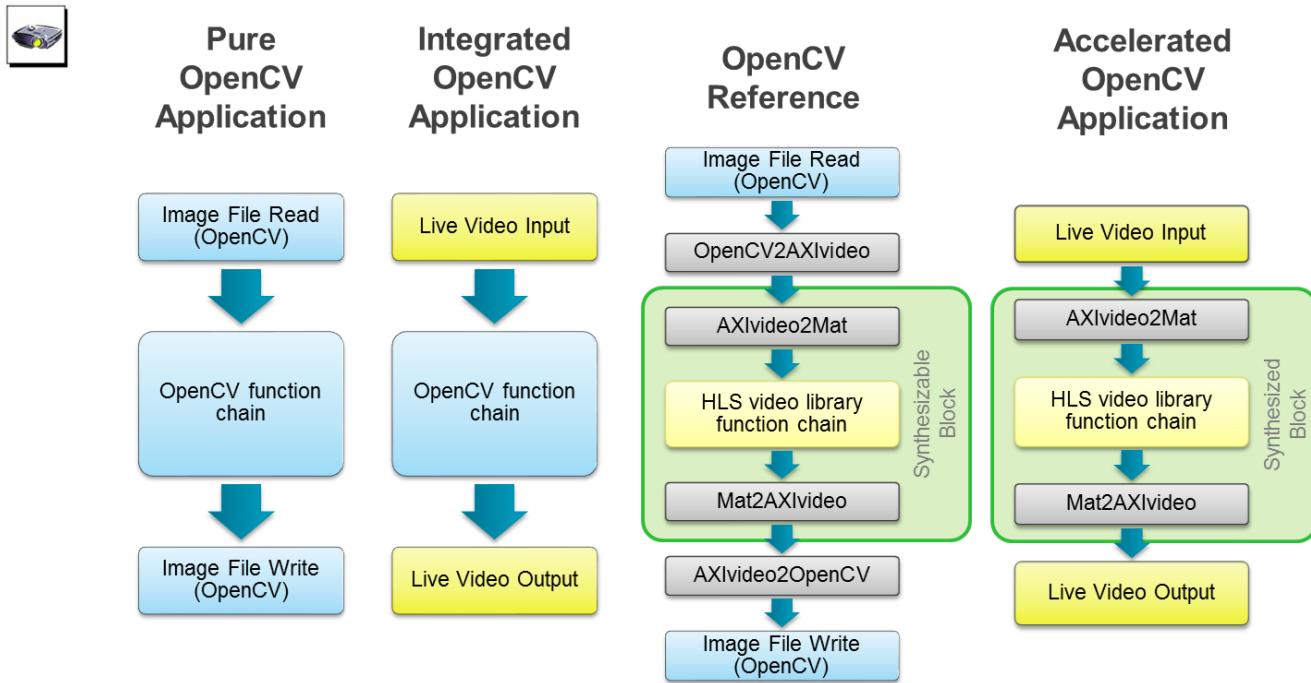


Video Data Modeling		AXI4-Stream IO Functions	
Linebuffer class	Window class	AXIvideo2Mat	Mat2AXIvideo
OpenCV Interface Functions			
<code>cvMat2AXIvideo</code>	<code>AXIvideo2cvMat</code>	<code>cvMat2hlsMat</code>	<code>hlsMat2cvMat</code>
<code>IplImage2AXIvideo</code>	<code>AXIvideo2IplImage</code>	<code>IplImage2hlsMat</code>	<code>hlsMat2IplImage</code>
<code>CvMat2AXIvideo</code>	<code>AXIvideo2CvMat</code>	<code>CvMat2hlsMat</code>	<code>hlsMat2CvMat</code>
Video Functions			
<code>AbsDiff</code>	<code>Duplicate</code>	<code>MaxS</code>	<code>Remap</code>
<code>AddS</code>	<code>EqualizeHist</code>	<code>Mean</code>	<code>Resize</code>
<code>AddWeighted</code>	<code>Erode</code>	<code>Merge</code>	<code>Scale</code>
<code>And</code>	<code>FASTX</code>	<code>Min</code>	<code>Set</code>
<code>Avg</code>	<code>Filter2D</code>	<code>MinMaxLoc</code>	<code>Sobel</code>
<code>AvgSdv</code>	<code>GaussianBlur</code>	<code>MinS</code>	<code>Split</code>
<code>Cmp</code>	<code>Harris</code>	<code>Mul</code>	<code>SubRS</code>
<code>CmpS</code>	<code>HoughLines2</code>	<code>Not</code>	<code>SubS</code>
<code>CornerHarris</code>	<code>Integral</code>	<code>PaintMask</code>	<code>Sum</code>
<code>CvtColor</code>	<code>InitUndistortRectifyMap</code>	<code>Range</code>	<code>Threshold</code>
<code>Dilate</code>	<code>Max</code>	<code>Reduce</code>	<code>Zero</code>

For function signatures and descriptions, see the HLS user guide

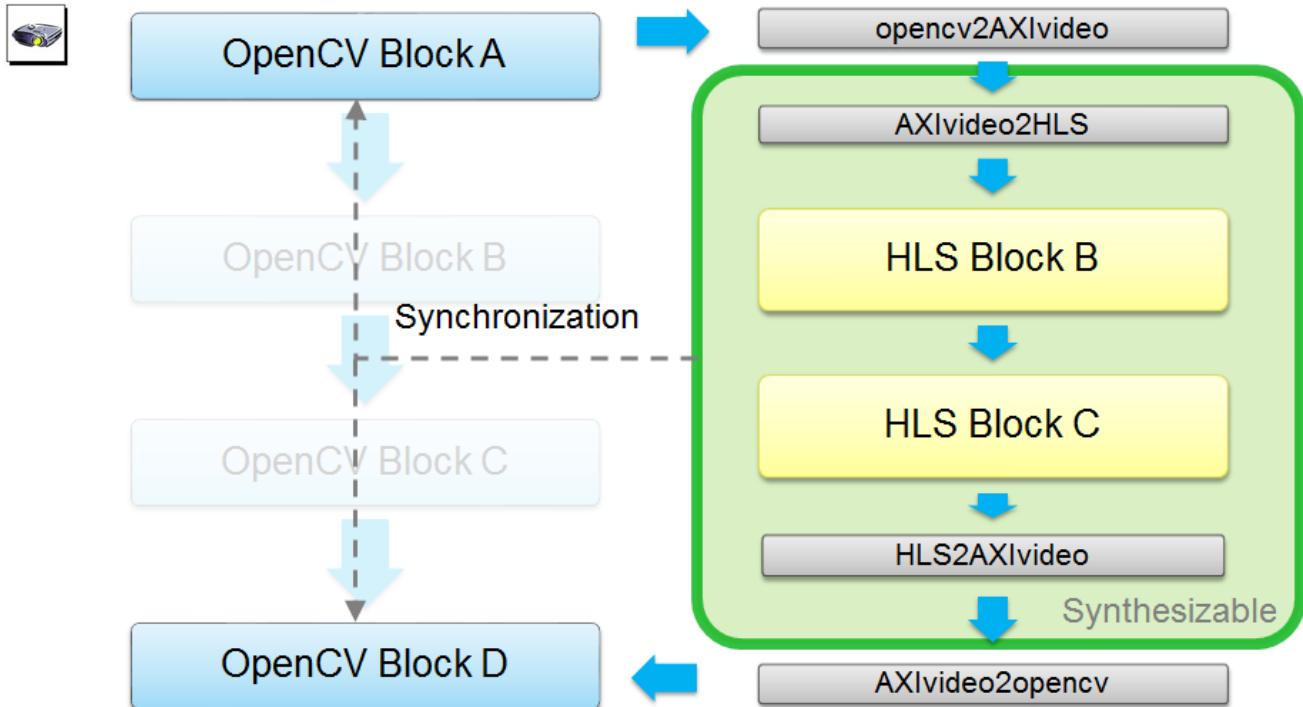
Slide 11-86:

Using OpenCV in FPGA Designs



Slide 11-87:

Partitioned OpenCV Application

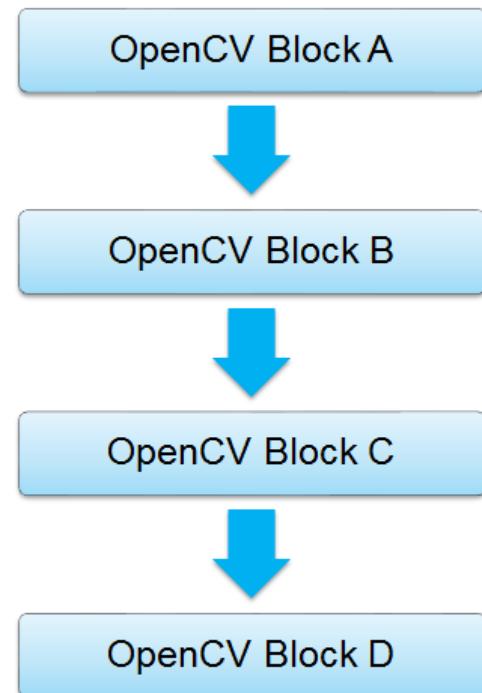


Slide 11-88:

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

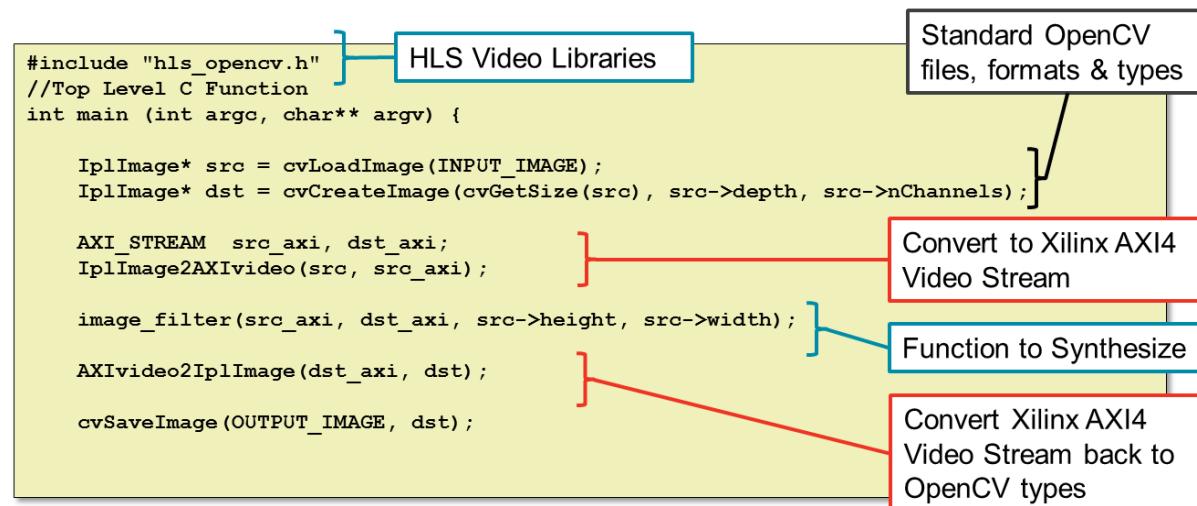


Slide 11-89:

C Testbench: Interface Library Example



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



Slide 11-90:

C Function to Synthesize 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

Slide 11-91:

Summary

- C Validation Flow
- C Language Support
- Data Types and Bit Accuracy
- Pointers
- Hardware Modeling
- OpenCV Libraries
- **Summary**



Slide 11-92:

Summary



- Overview of Vivado HLS tool language support
 - Data types and bit accuracy
 - Arbitrary precision types can define bit-accurate operators leading to better QoR
 - C validation flow
 - Validate the function before synthesis
 - Use a self-checking testbench that returns 0
 - C language support
 - C, C++, and SystemC are supported
 - Pointers and streams
 - Be aware of some modeling issues when using pointers
 - Stream and shift registers classes may better model and implement hardware
 - HLS video libraries
 - Drop-in replacement for OpenCV and provide high QoR

Capture Your Notes Here



Appendix: I/O Interfaces

Slide 12-1:

2016.1

This module is an appendix to the "I/O Interfaces" module.

Slide 12-2:

Vivado HLS Tool I/O Ports, Protocols, and Options



- **Vivado HLS Tool I/O Ports, Protocols, and Options**
- Port-Level I/O Protocols: Wire Handshakes
- Coding Issues and I/O

Slide 12-3:

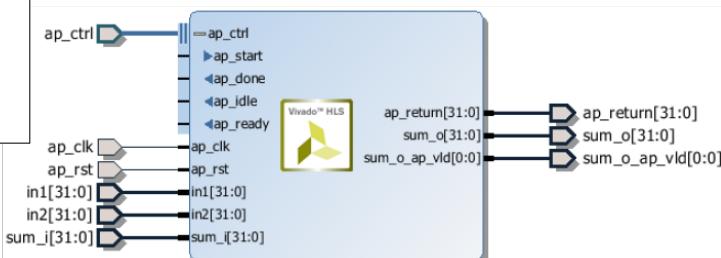
Interface Synthesis Overview



- Function arguments become input/output (port) of RT
- The following code provides an overview of interface synthesis

```
#include "sum_io.h"

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;
}
```





The example above includes:

- Two pass-by-value inputs `in1` and `in2`
- A pointer sum that is both read from and written to
- A function return, the value of `temp`

With the default interface synthesis settings, the design is synthesized into an RTL block with the ports shown in the above figure.

The Vivado HLS tool creates three types of ports in the RTL design:

- Clock and reset ports: `ap_clk` and `ap_rst`
- Block-level interface protocol:
 - `ap_start`, `ap_done`, `ap_ready`, and `ap_idle`
- Port-level interface protocols. These are created for each argument in the top-level function and the function return (if the function returns a value). In this example, these ports are:
 - `in1`, `in2`, `sum_i`, `sum_o`, `sum_o_ap_vld`, and `ap_return`.

Slide 12-4:

Vivado HLS Tool I/O Options



- Data ports
 - Directly derived from the function arguments / parameters
- Block-level interfaces (optional)
 - An interface protocol that is added at the block level
 - Controls the addition of block level control ports: start, idle, done, and ready
- Port-level interfaces (optional)
 - I/O interface protocols added to the individual function arguments
- Bus interfaces (optional)
 - Added as external adapters when the RTL is exported as an IP block

Slide 12-5:

Vivado HLS Tool I/O Options: Basic Ports

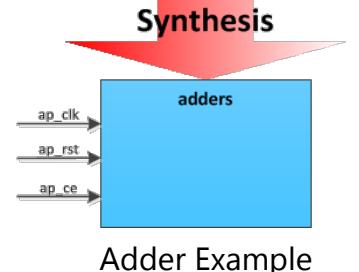


- Clock added to all RTL blocks
 - One clock per function / block
 - SystemC designs may have a unique clock for each CTHREAD

- Reset added to all RTL blocks
 - Only the FSM and any variables initialized in the C are reset by default
 - Reset and polarity options are controlled via the RTL configuration
 - Solutions / Solution Settings

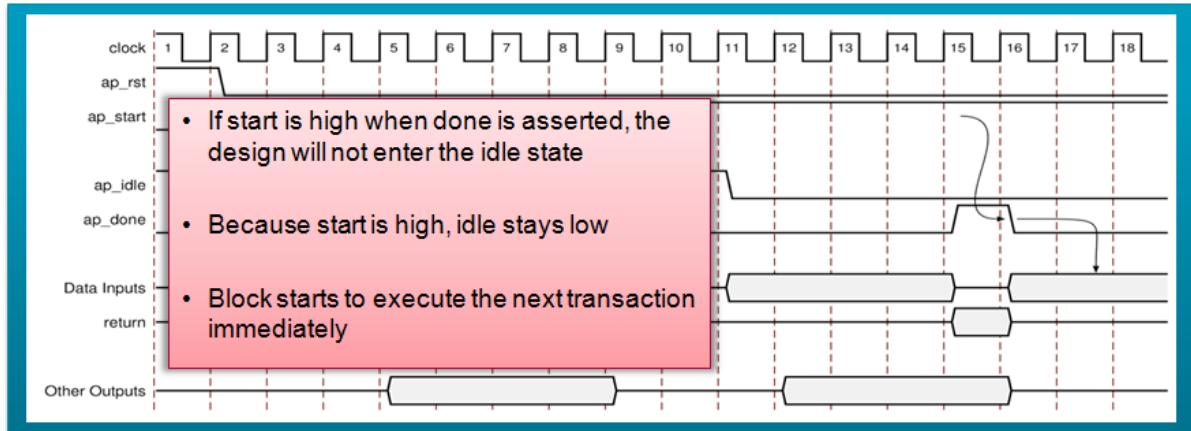
- Optional clock enable
 - An optional clock enable can be added via the RTL configuration
 - When de-asserted it will cause the block to freeze
 - All connected blocks are assumed to be using the same CE
 - When the I/O protocol of this block freezes, it is expected other blocks will do the same
 - Otherwise a valid output may be read multiple times

```
#include "adders.h"
int adders(int in1, int in2, int *sum) {
    int temp;
    *sum = in1 + in2 + *sum;
    temp = in1 + in2;
    return temp;
}
```



Slide 12-6:

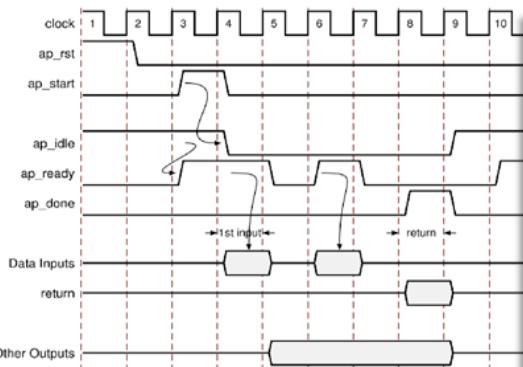
AP_START: Constant High



- Input and output data operations
 - As before
- Key difference here is that the design is never idle
 - Next data read is performed immediately

Slide 12-7:

Pipelined Designs



- After reset, ap_ready goes high if idle is high
- Start applied: Idle goes low, apply first data
- Inputs will be read when ap_ready is high and idle is low
- Output ap_done goes high on the final cycle
- Idle goes high one cycle after done
- Block waits for next start

This example shows an II of 2

■ Input data

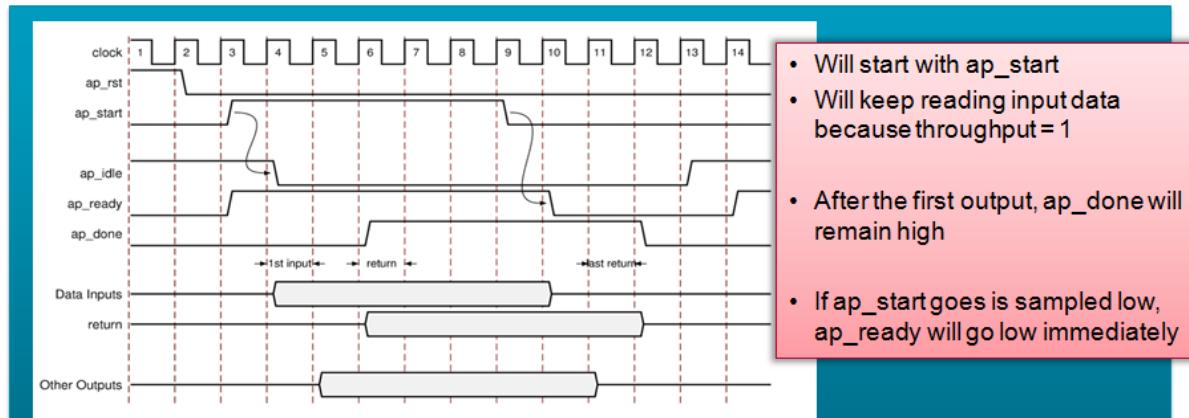
- Will be read when ap_ready is high and ap_idle is low
 - Indicates the design is ready for data
- Signal ap_ready will change at the rate of the throughput

■ Output data

- As before, function return is valid when ap_done is asserted high
- Other outputs may output their data at any time after the first read
 - Using a port-level I/O protocol for other outputs is recommended

Slide 12-8:

Pipelined Designs: Throughput = 1



■ Input data when TP=1

- It can be expected that `ap_ready` remains high and data is continuously read
- Design will only stop processing when `ap_start` is de-asserted

■ Output data when TP=1

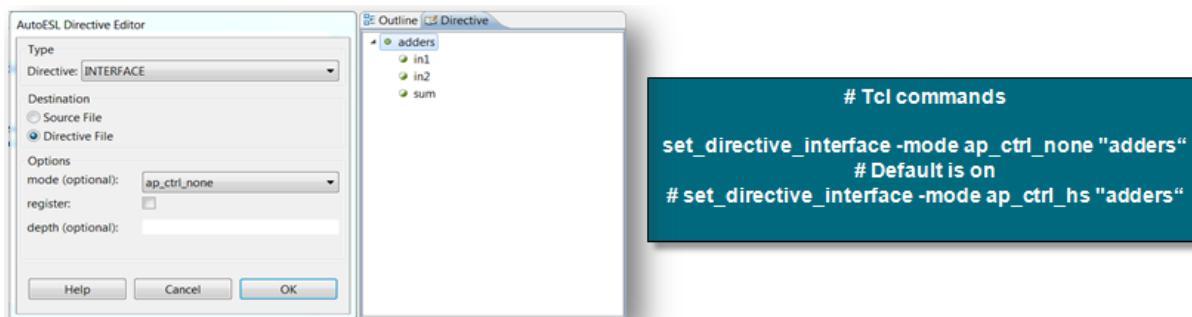
- After the first output, `ap_done` will remain high while there are samples to process
 - Assuming there is no data decimation (output rate = input rate)

Slide 12-9:

Disabling Block-Level Handshakes



- Block-level handshakes can be disabled
 - Select the function in the directives tab, right-click
 - Select **Interface** and then **ap_ctrl_none** for no block-level handshakes
 - Select **Interface** and then **ap_ctrl_hs** to re-apply the default



- New requirement: Manually verify the RTL
 - Without block-level handshakes, C/RTL co-simulation cannot verify the design
 - Will only work in simple combo and TP1 cases
 - Handshakes are required to know when to sample output signals

Leaving the default and using block-level handshakes is recommended

Slide 12-10:

Block-Level Handshake Summary

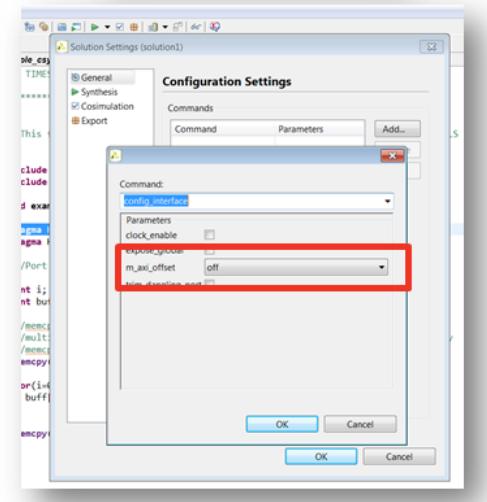
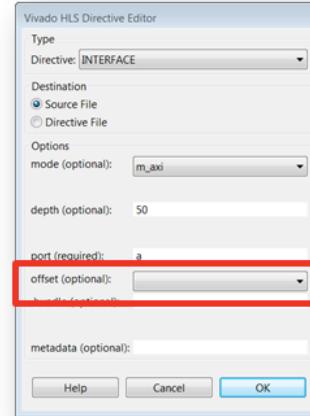


- Block-level handshakes are added to the RTL design
 - These are in addition to any data arguments
 - These signals help to interface the RTL block with other blocks
 - Can be optionally disabled
- Enables system-level control and sequencing
 - Only indication (ap_done) that the function return (ap_return) is valid for reading
 - Only way to **prevent** data being processed (make ap_start=0)
 - Only overhead is an additional clock cycle at the beginning to sample the start signal
 - Can be tied high and this overhead will only be incurred once, after reset
- Recommendation
 - Making use of block-level handshakes is safer and more productive

Slide 12-11:

AXI4-Master: Offset Support

- Address offset / base address support
 - Support provided for the address offset
- Port offset
 - Defines the offset for the port
 - Can be set on individual interfaces using the INTERFACE directive
- Global offset
 - Globally controls the offset ports of all M_AXI interfaces in the design
 - Can be set using the interface configuration
 - Using the Tcl command
config_interface
-m_axi_offset



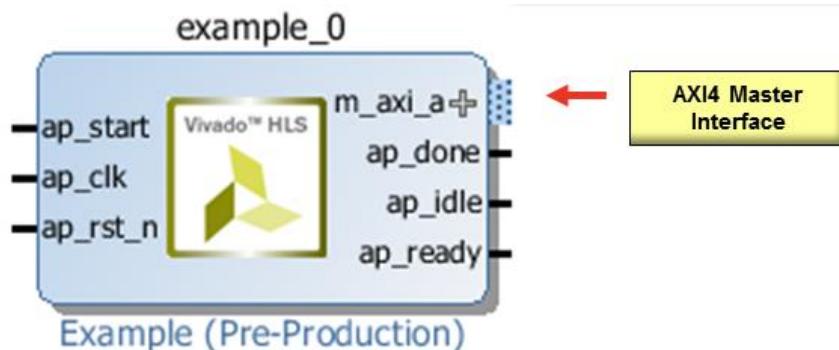
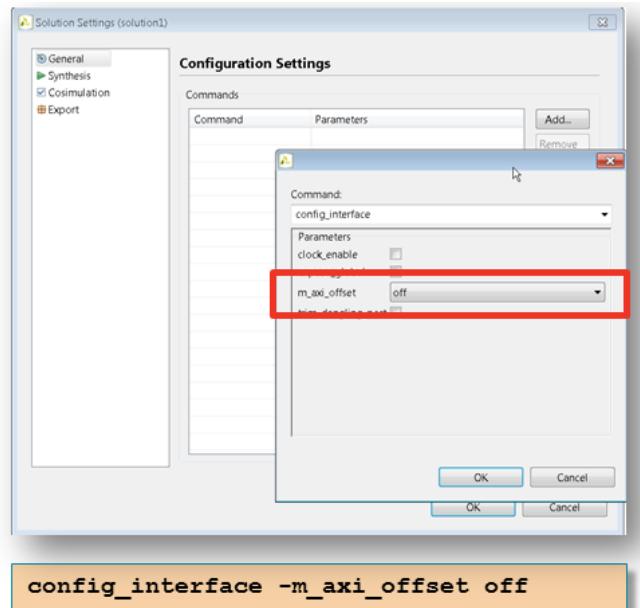
Slide 12-12:

AXI4-Master: Offset=off (default)



- Default AXI4-Master interface
 - No offset is provided for the address

- The offset (BASEADDR) is set in IPI
 - Using the IP customization GUI
 - Offset cannot be changed on the fly

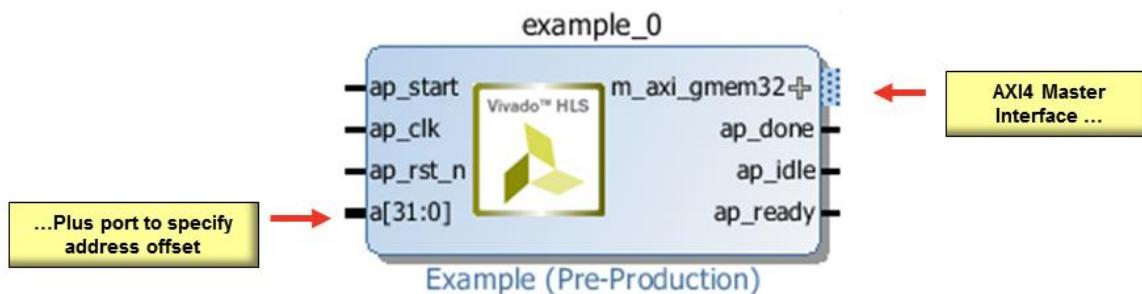
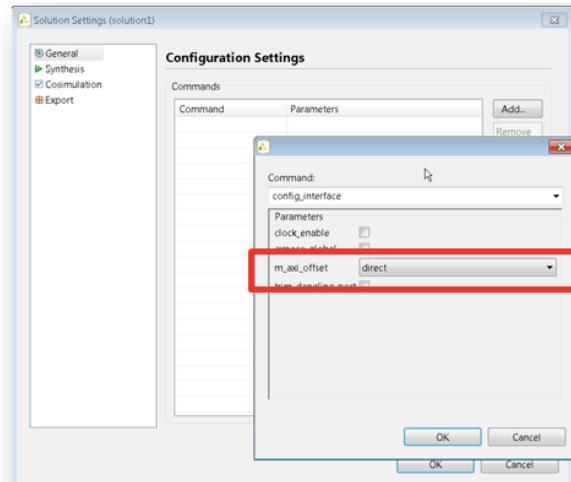


Slide 12-13:

AXI4-Master: Offset=direct



- Direct interface
 - Generates a scalar input offset port
 - The offset is set by driving the input port
 - It can be changed on the fly by driving the port with a different value

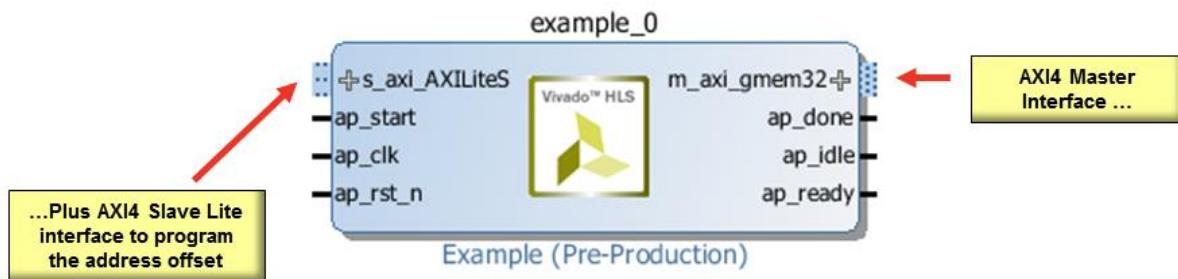
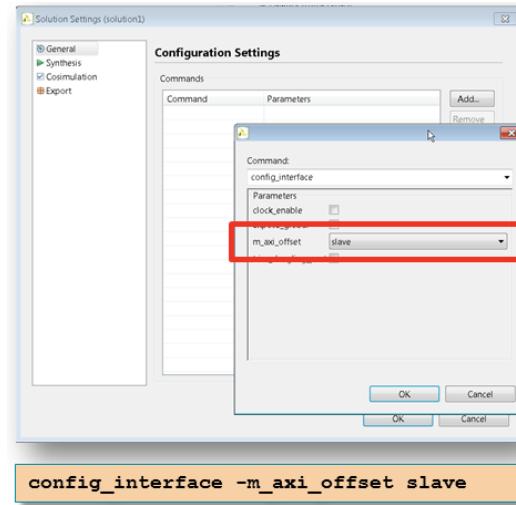


Slide 12-14:

AXI4-Master: Offset=slave



- Direct interface
 - Generates an offset port and automatically maps it to an AXI4-Slave Lite interface
 - User must program the offset before starting transactions on the AXI4-Master interface
 - It can be changed on the fly by re-programming the offset register



Slide 12-15:

Port-Level I/O Protocols: Wire Handshakes



- Vivado HLS Tool I/O Ports, Protocols, and Options
- **Port-Level I/O Protocols: Wire Handshakes**
- Coding Issues and I/O

Slide 12-16:

Wire Protocols (1)



- Wire protocols add a valid and/or acknowledge port to each data port
- Wire protocols are all derivatives of protocol ap_hs
 - ap_ack: add an acknowledge port
 - ap_vld: add a valid port
 - ap_ovld: add a valid port to an output
 - ap_hs: adds both

Slide 12-17:

Wire Protocols (2)



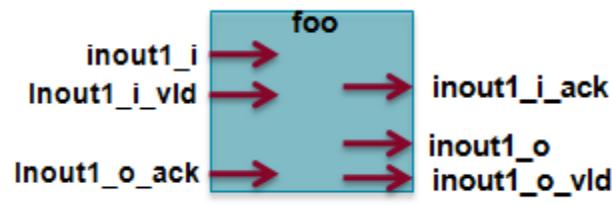
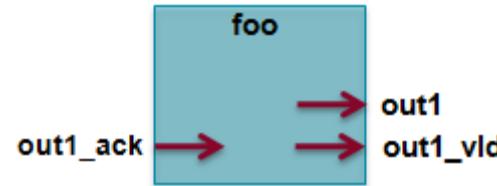
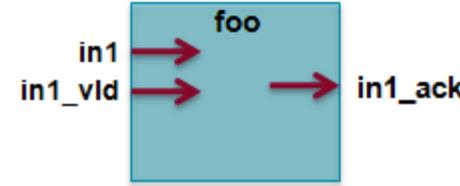
- Output control signals are used to inform other blocks
 - Data has been read at the input by this block (ack)
 - Data is valid at the output (vld)
 - Other block must accept the control signal (this block will continue)
- Input control signals are used to inform this block
 - Output data has been read by the consumer (ack)
 - Onput from the producer is valid (vld)
 - **This block will stall** while waiting for the input controls

Slide 12-18:

Wire Protocols: Ports Generated

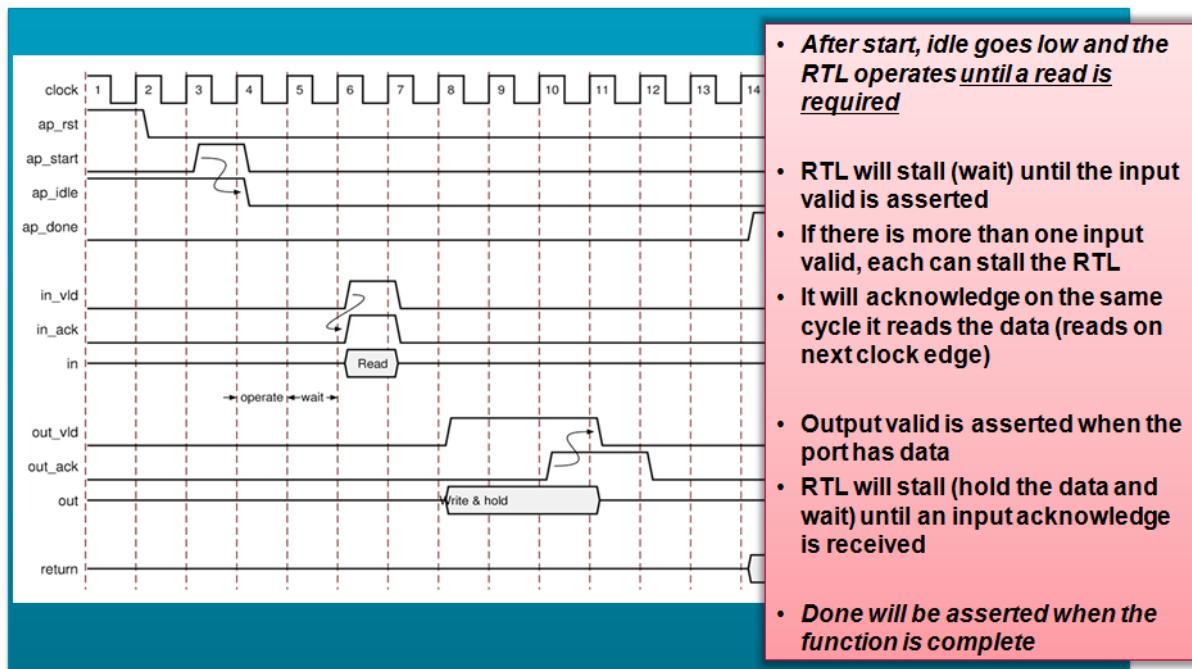


- Wire protocols are all derivatives of protocol ap_hs
 - Inputs
 - Arguments that are only read
 - Valid is input port indicating when to read
 - Acknowledge is an output indicating it was read
 - Outputs
 - Arguments that are only written to
 - Valid is an output indicating data is ready
 - Acknowledge is an input indicating it was read
 - Inouts
 - Arguments that are read from and written to
 - These are split into separate in and out ports
 - Each half has handshakes as per input and output



Slide 12-19:

Handshake I/O Protocol



Slide 12-20:

Other Handshake Protocols (1)



- Other wire protocols are derivatives of ap_hs in behavior
 - Some subtleties are worth discussing when the full two-way handshake is **not** used
- Using the valid protocols (ap_vld, ap_ovld)
 - **Outputs:** Without an associated input acknowledge it is a requirement that the consumer takes the data when the output valid is asserted
 - Protocol ap_ovld only applies to output ports (is ignored on inputs)
 - **Inputs:** No particular issue
 - Without an associated output acknowledge, the producer does not know when the data has been read (but the done signal can be used to update values)

Slide 12-21:

Other Handshake Protocols (2)



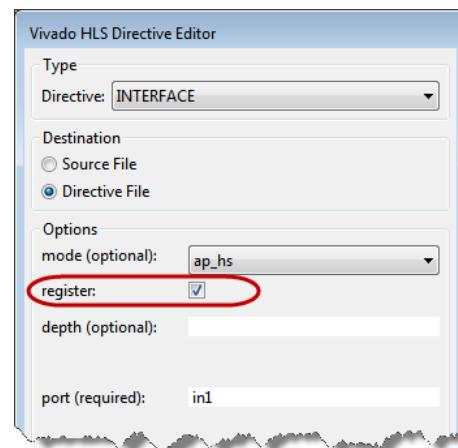
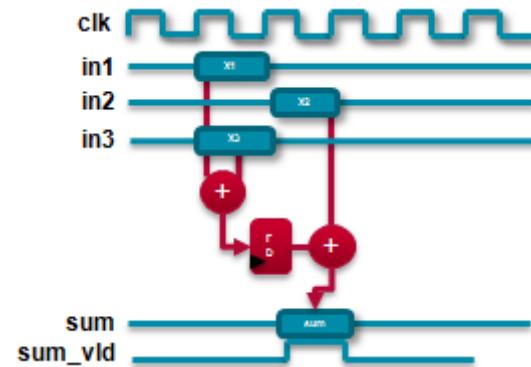
- Using the acknowledge protocol (ap_ack)
 - **Outputs:** Without an associated output valid the consumer will not know when valid data is ready but the design will stall until it receives an acknowledge
 - **Dangerous: lock up potential**
 - **Inputs:** Without an associated input valid, the design will simply read when it is ready and acknowledge that fact

Slide 12-22:

Registering I/O Reads and Writes



- Vivado HLS tool does not register input and outputs by default
 - Will chain operations to minimize latency
 - Inputs will be read when the design requires them
 - Outputs will be written as soon as they are available
- Inputs and outputs can be registered
 - Inputs will be registered in the first cycle
 - Input pointers and partitioned arrays will be registered when they are required
 - Outputs will be registered and held until the next write operation
 - Which for scalars will be the next transaction (cannot write twice to the same port) unless the block is pipelined



Slide 12-23:

Wire Handshake Summary



- Wire handshakes are flexible and the defaults
 - Can be used for all C arguments except arrays
 - Default for all C arguments except arrays
 - Inputs default to none, outputs default to output valid
- Input valid and acknowledge signals will stall the RTL
 - RTL will wait until an input is valid
 - After writing an output to the port, it will wait for the output to be acknowledged
 - Any port can stall the designs: each can block
- Recommendation
 - Making use of full two-way handshake is safer and more productive
 - Also similar to AXI4-Stream
 - Be very careful about using an acknowledge only protocol on output ports

Slide 12-24:

Coding Issues and I/O



- Vivado HLS Tool I/O Ports, Protocols, and Options
- Port-Level I/O Protocols: Wire Handshakes
- **Coding Issues and I/O**

Slide 12-25:

Coding and I/O



- One major issue with I/O and coding
 - Multi-access pointers
- Multi-access pointer is a pointer argument that 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;
        }
    }
}
```

The diagram shows a C code snippet for a FIFO function. It includes a blue bracket on the left side of the loop body, with two arrows pointing from it to two yellow callout boxes. The top box contains the text "*d_i is read multiple times" and the bottom box contains the text "*d_o is written to multiple times".

- This can result in incorrect hardware
- This cannot be verified with a testbench
- This can give incorrect results at RTL verification

Yes, you can use this coding style (if you REALLY want)...with the following workarounds

Slide 12-26:

Multi-Access Pointers: The Effect of Volatile

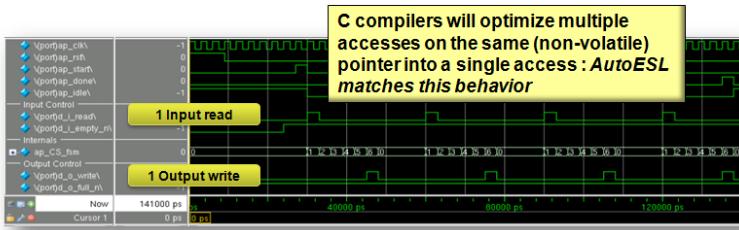
- Easier to see with a simple example



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

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

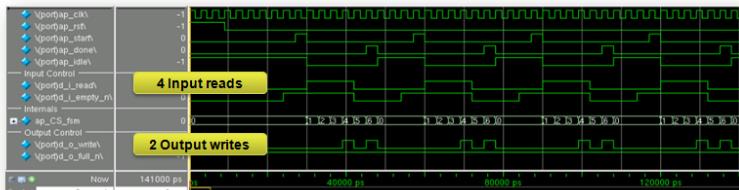
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```



```
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;
}
```



Slide 12-27:

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;
        }
    }
}
```

- 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 optimize to 1 read and 1 write
- Vivado HLS tool will create a design with 1 read and 1 write

- C compilers will not optimize the I/O accesses
 - Will assume the pointer value may change outside the scope of the function and leave the accesses
- Vivado HLS tool will create a design with 4 reads and 2 writes

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

Slide 12-28:

Multi-Access Pointers: Limited Visibility



- Intermediate results are hard to use for verification
 - Only the final pointer value is passed to the testbench
 - Final value of *d_o can be captured in the testbench and compared
 - Intermediate values can only be seen by using printf: they cannot be automatically verified by the testbench

```
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;
        }
    }
}
```

- 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 can be used instead

Slide 12-29:

Multi-Access Pointers: Testbench 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;
    }
}
```

C testbench may correctly provide four values

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

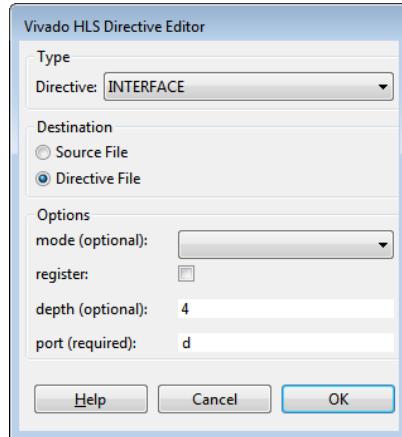
    for (i=0;i<4;i++) {
        d[i] = i;
    }

    foo(d);

    return 0;
}
```

- AutoSim (RTL verification) only sees a pointer on the interface

- And will create a SystemC co-simulation wrapper to supply or store one sample
- Use the **depth** option to specify the actual depth
- Vivado HLS tool 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 mismatch

Capture Your Notes Here



HLS UltraFast Design Methodology

Slide 13-1:

2016.1



This module covers the HLS UltraFast™ design methodology recommendations to obtain the required performance quickly. 30 minutes.

Slide 13-2:

HLS UltraFast Design Methodology



Simulate Design

- Validate the C function

Synthesize Design

- Baseline design

1: Initial Optimizations

- Define interfaces (and data packing)
- Define loop trip counts

2: Pipeline for Performance

- Pipeline and dataflow

3: Optimize Structures for Performance

- Partition memories and ports
- Remove false dependencies

4: Reduce Latency

- Optionally specify latency requirements

5: Improve Area

- Optionally recover resources through sharing



- Synthesize the initial design
 - Ensure C code can be synthesized and get the baseline performance
 - Then follow the steps in the HLS UltraFast design methodology

1) Initial optimizations

- Define interfaces (and data packing), define loop trip counts

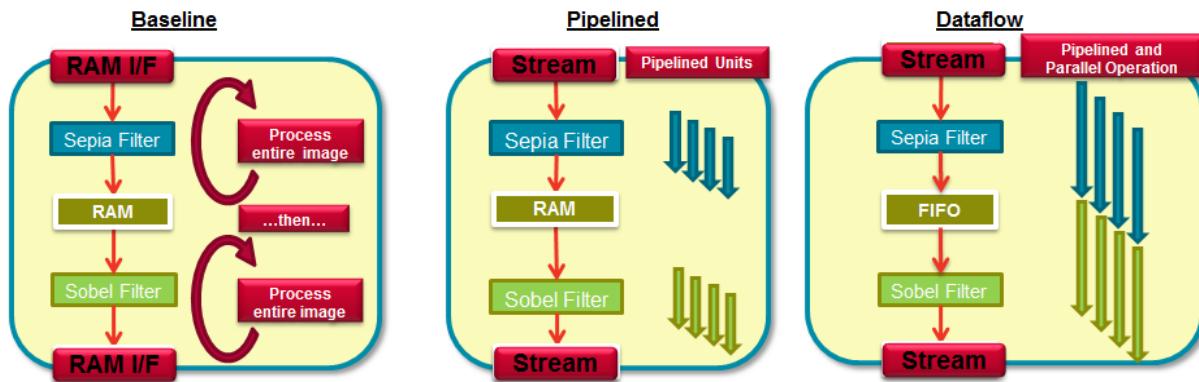
- 2) Pipeline for Performance
 - Pipeline and dataflow loops, functions and operations
- 3) Optimize Structures for Performance
 - Partition block RAMs and ports; remove false dependencies
- 4) Reduce Latency
 - Optionally reduce the latest to get results earlier (may increase area)
- 5) Improve Area
 - Optionally recover resources by improving sharing

Slide 13-3:

Example: Design and Performance Improvement



- Performance improvement using pipeline with dataflow
 - From baseline to using pipelined tasks to using dataflow



	Baseline	Pipelined	Dataflow
BRAM	2792	2790	24
FF	891	1136	883
LUT	2315	2114	1606
Interval	128,744,588	4,150,224	2,076,613

Slide 13-4:

Step 1: Initial Optimizations – Directives



Directives and Configurations	Description
INTERFACE	Specifies how RTL ports are created from the function description.
DATA_PACK	Packs the data fields of a struct into a single scalar with a wider word width.
LOOP_TRIPCOUNT	Used for loops that have variable bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.
Config Interface	This configuration controls I/O ports not associated with the top-level function arguments and allows unused ports to be eliminated from the final RTL.



The design interface is typically defined by the other blocks in the system. Since the type of I/O protocol helps determine what can be achieved by synthesis, it is recommended that the INTERFACE directive be used to specify this before proceeding to optimize the design.

If the algorithm accesses data in a streaming manner, you might want to consider using one of the streaming protocols to ensure high performance operation.

Tip: If the I/O protocol is completely fixed by the external blocks and will never change, consider inserting the INTERFACE directives directly into the C code as pragmas.

When structs are used in the top-level argument list, they are decomposed into separate elements and each element of the struct is implemented as a separate port. In some cases, it is useful to use the DATA_PACK optimization to implement the entire struct as a single data word, resulting in a single RTL port.

Care should be taken if the struct contains large arrays. Each element of the array is implemented in the data word and this might result in a very wide data ports.

A common issue when designs are first synthesized is report files showing the latency and interval as a question mark "?" rather than as numerical values. If the design has loops with variable loop bounds, the Vivado® HLS tool cannot determine the latency and uses the "?" to indicate this condition.

To resolve this condition, use the analysis perspective or the synthesis report to locate the lowest level loop for which synthesis fails to report a numerical value and use the LOOP_TRIPCOUNT directive to apply an estimated tripcount. This allows values for latency and interval to be reported and allows solutions with different optimizations to be compared.

Finally, global variables are generally written to and read from, within the scope of the function for synthesis and are not required to be I/O ports in the final RTL design. If a global variable is used to bring information into or out of the C function, you might want to expose them as an I/O port using the interface configuration.

Slide 13-5:

Step 2: Pipeline for Performance

Directives and Configurations	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.
DATAFLOW	Enables task-level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval.
RESOURCE	Specifies a resource (core) to use to implement a variable (array, arithmetic operation, or function argument) in the RTL.
Config Compile	Allows loops to be automatically pipelined based on their iteration count.



At this stage of the optimization process you want to create as much concurrent operation as possible. You can apply the PIPELINE directive to functions and loops. You can also use the DATAFLOW directive at the level that contains the functions and loops to make them work in parallel.

A recommended strategy is to work from the bottom up and be aware of the following:

- Some functions and loops contain sub-functions. If the sub-function is not pipelined, the function above it might show limited improvement when it is pipelined. The non-pipelined sub-function will be the limiting factor.

- Some functions and loops contain sub-loops. When you use the PIPELINE directive, the directive automatically unrolls all loops in the hierarchy below. This can create a great deal of logic. It might make more sense to pipeline the loops in the hierarchy below.
- Loops with variable bounds cannot be unrolled, and any loops and functions in the hierarchy above these loops cannot be pipelined. To address this issue, pipeline these loops, and use DATAFLOW optimization to maximize the performance of the function that contains the loop. Alternatively, rewrite the loop to remove the variable bound.

Slide 13-6:

Loops with Variable Limits



- Variable-bound loops
 - Typically due to the iteration limit being an input variable
 - Vivado® HLS tool cannot determine the loop iteration limit: it is variable
- Pipeline cannot auto-unroll
 - Vivado HLS tool cannot unrolls these loops
 - In the example, L1 cannot be pipelined because L2 cannot be auto-unrolled
 - Recode to allow unrolling
- No latency reported
 - Vivado HLS tool cannot know the number of loop iterations
 - It therefore cannot report the total latency of loops or design

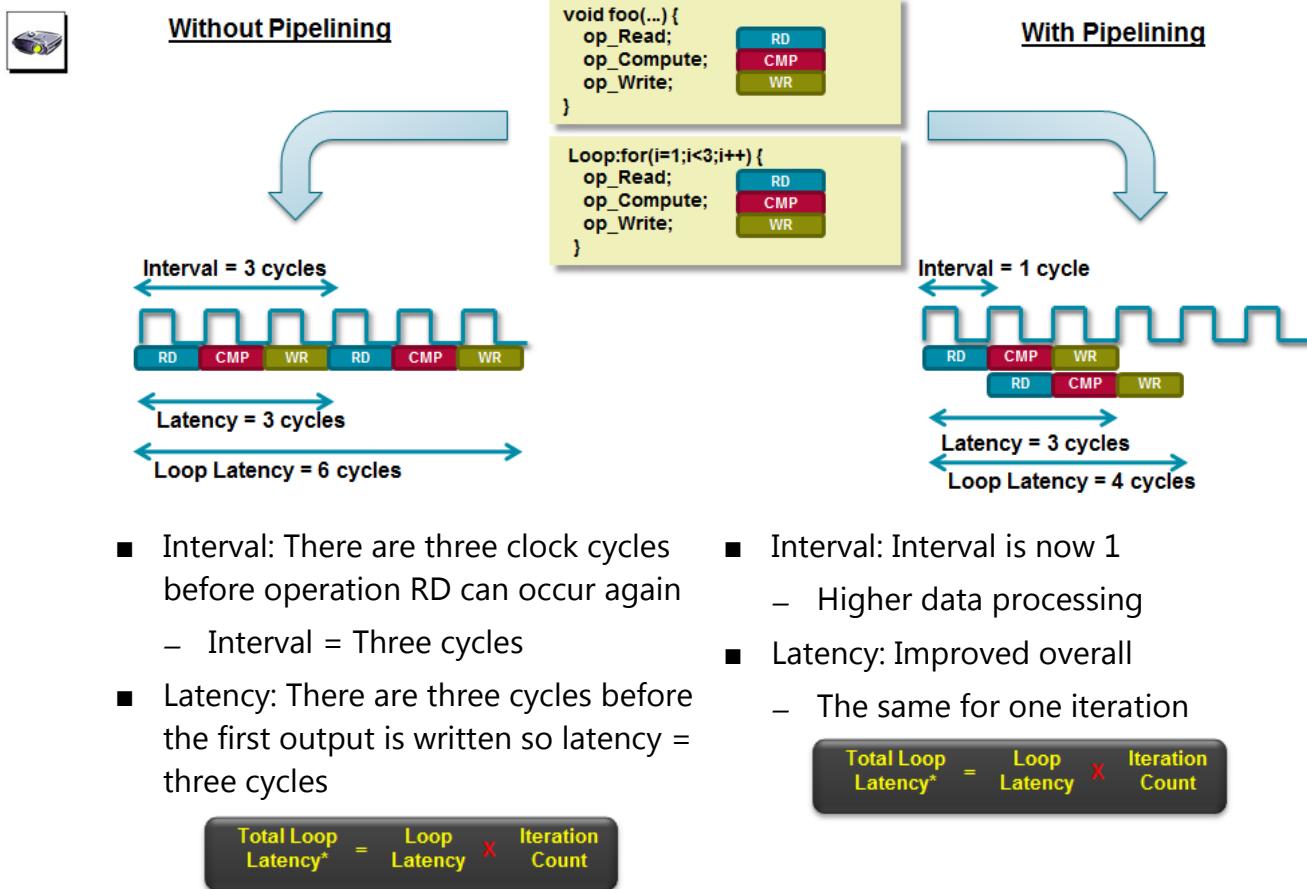
```
void foo(in1[N][M], in2[N][M], Rows,
        Cols...) {
...
    L1:for(i=1;i<Rows;i++) {
        #pragma HLS PIPELINE
        L2:for(j=0;j<Cols;j++) {
            out[i][j] = in1[i][j] + in2[i][j];
        }
    }
}
```



```
void foo(in1[N][M], in2[N][M], Rows,
        Cols...) {
...
    L1:for(i=1;i<Rows;i++) {
        #pragma HLS PIPELINE
        N = max_size_of_Cols
        L2:for(j=0;j<N;j++) {
            if (j >= N)
                break;
            out[i][j] = in1[i][j] + in2[i][j];
        }
    }
}
```

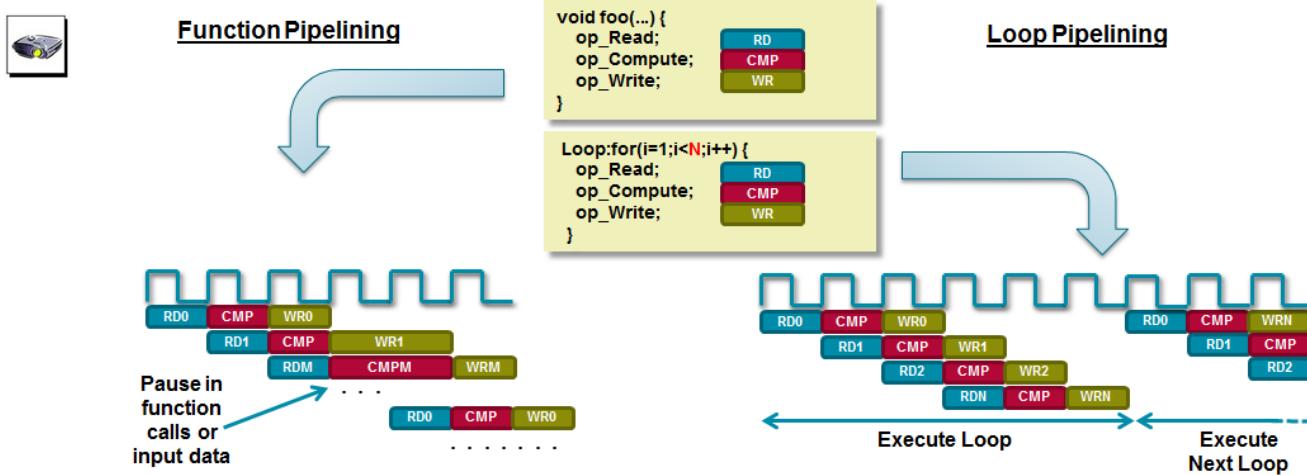
Slide 13-7:

Review: Loop and Function Pipelining



Slide 13-8:

Loop Versus Function Pipelining



- | | |
|---|---|
| <ul style="list-style-type: none"> Executed each time the function is called | <ul style="list-style-type: none"> Will always execute N iterations <ul style="list-style-type: none"> N is the loop iteration count or trip count |
|---|---|

Three topics to review:

- I/O behavior
- Stall behavior
- Flush behavior

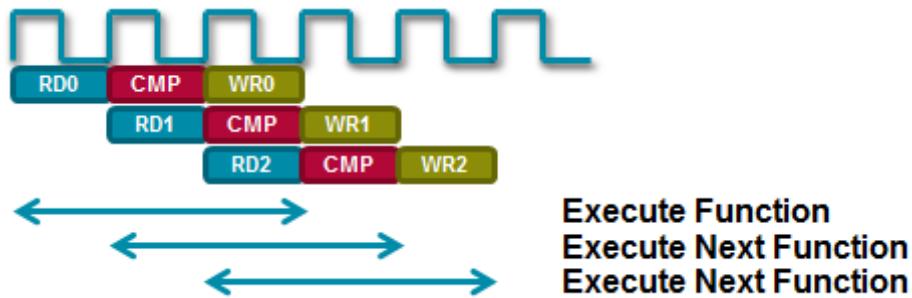
Slide 13-9:

Function Pipelining: I/O Behavior



- Pipelined function
 - Executes each time a new input sample is received
 - Data is not flushed by default

```
void foo(...) {
    op_Read;
    op_Compute;
    op_Write;
}
```



- The function can always read and write
 - No inherent bubble in the C description



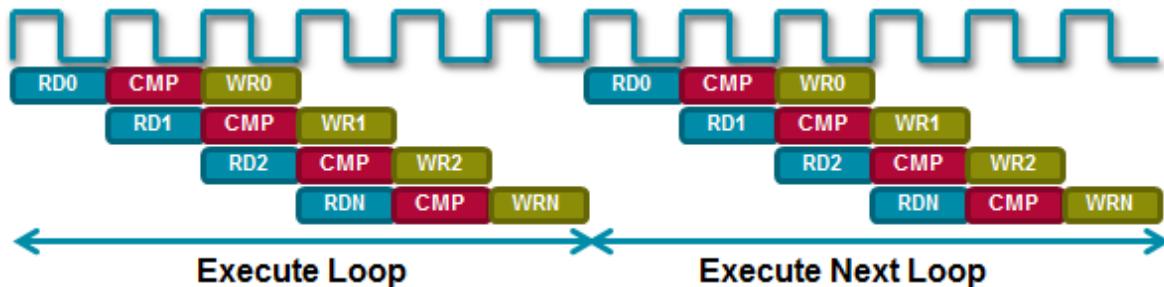
Slide 13-10:

Loop Pipelining: I/O Behavior

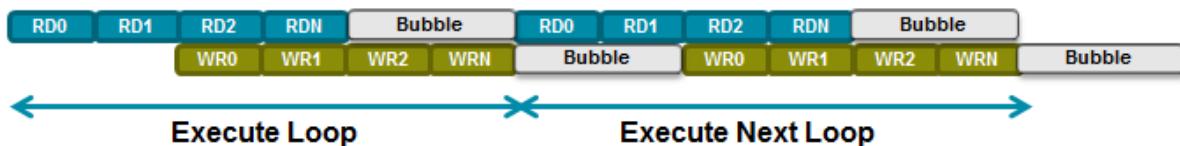


- Executes all loop iterations
 - Loops execute until all loop iterations are complete

```
Loop: for (i=1; i<N; i++) {
    op_Read;
    op_Compute;
    op_Write;
}
```



- Loop stops reading and writing between iteration
 - Reads for the next loop will not start until current loop ends
 - Cannot read every sample due to a bubble inherent in the loop behavior

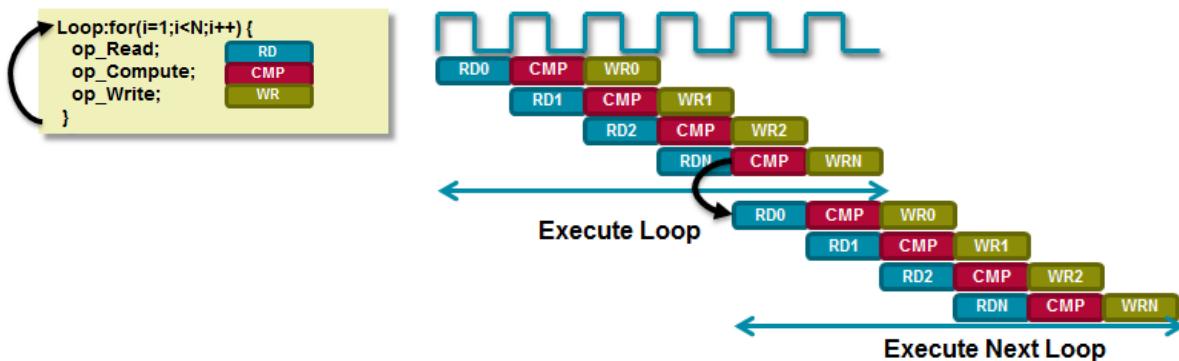


Slide 13-11:

Loop Rewind



- Optional pipeline implementation
 - Use the rewind option
- Causes the loop to rewind and execute immediately
 - As soon as the next input operation can begin



- Can only be applied
 - To the top-level loop in a function
 - To a loop used in a dataflow region

Slide 13-12:

When to Use Loop Rewind?



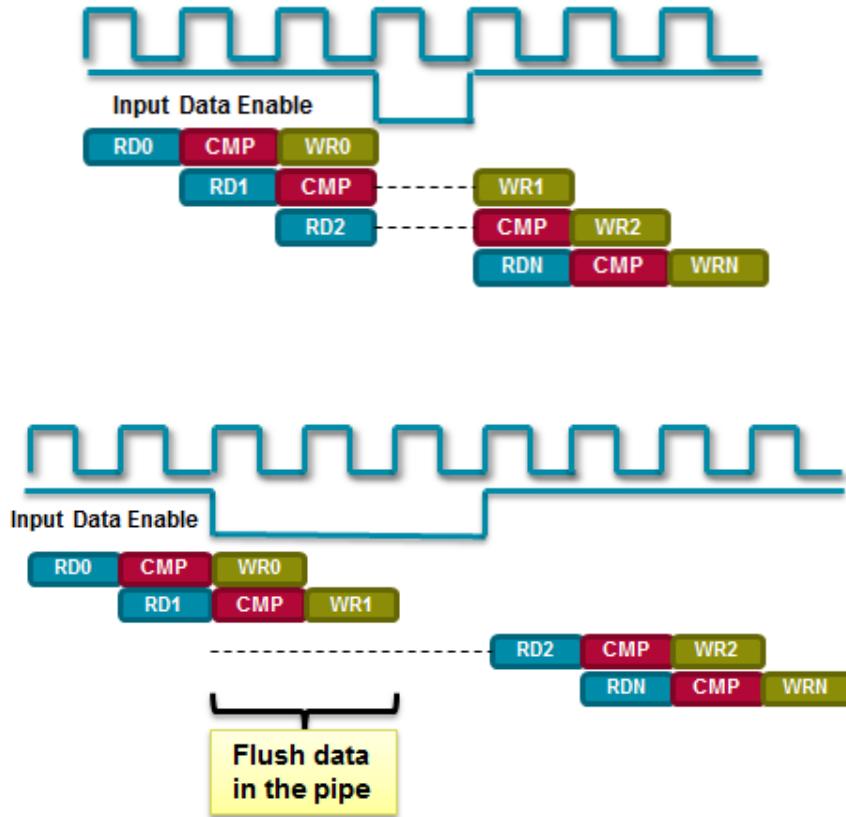
- To meet the cycle budget
 - Loops without rewind will introduce "bubbles" in the data
 - This will increase the interval and lengthen the latency
 - More cycles to process all samples
 - If the interval is greater than the cycle budget, rewind loops to improve performance
- To satisfy system requirements
 - If the system cannot tolerate "bubbles" in the data stream
 - Rewind loops to ensure continuous processing of the data

Slide 13-13:

Function Stall and Flush Behavior



- No input data
 - If no input data available, the function will stall
 - Wait for next data, then continue
- Pipeline flush
 - Option for function pipelines
 - Function will flush data
- Functions
 - Process continuously
 - Do not flush data by default



Slide 13-14:

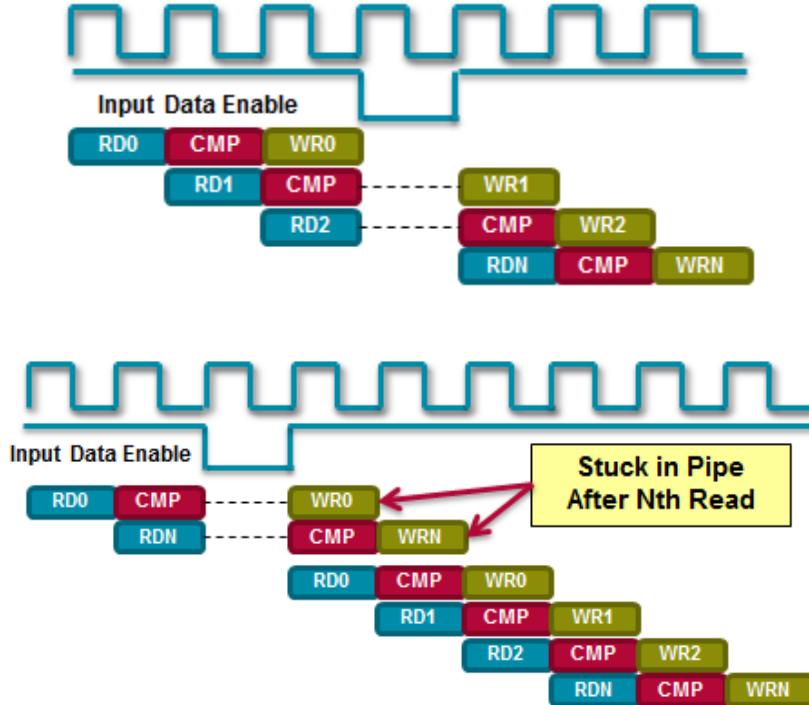
Loop Stall and Flush Behavior



- Loops can stall
 - If no input data available the loop will stall
 - Wait for next data, then continue

- Loops require input data
 - If loop is given N inputs, then stalls
 - Data will stall in the pipe
 - Will flush only after L new data samples
 - L is the latency of the pipeline

- Loops with rewind
 - Process continuously
 - Cannot flush data



Slide 13-15:

Loop Versus Function Pipelining Summary



	Loop	Function
Can data be processed continuously?	No. Rewind must be used on a top-level loop or on a loop in a dataflow region.	Yes.
Can data be stalled?	Yes.	Yes.
Can data be flushed?	No. Data can only be pushed out by new incoming data.	Yes.

- Now that you understand how these tasks can process pipelined data
 - Which loops and functions should be pipelined?

Slide 13-16:

C Code Description: By Sample or By Frame



- Two ways to capture any C function behavior
 - The following applies to the top-level function AND any other function in the hierarchy



- The input/output is an array
 - There may be scalars for control
- A frame of data is processed
 - Loops will typically process each set of data
 - Sub-functions may process data
- The input/output is a single sample
 - There may be arrays for control
- A sample of data is processed
 - Functions will typically process each set of data
 - Loops may process data
 - Either at the bit level or
 - As history in a shift register

Slide 13-17:

Frame Based: Pipeline the Sample Loop



- Sample loop
 - Loop that operates on data samples
 - In this case, this is loop L2
 - Pipelining this loop will allow the function to process one sample per clock
 - Rewind required for continuous processing
- The inner loop may not be the sample loop
 - Sub-loops can manipulate bit-level operations
 - Or perform multiple calculations per clock
 - These **must** be unrolled
 - Else the design will process 1-bit level operation per clock cycle
- Pipeline the functions

```
void foo(in1[N][M], in2[N][M], ...) {  
    ...  
    L1:for(i=1;i<N;i++) {  
        L2:for(j=0;j<M;j++) {  
            #pragma HLS PIPELINE  
            out[i][j] = in1[i][j] + in2[i][j];  
        }  
    }  
}
```

```
void image(in1[N][M], in2[N][M], Rows, Cols) {  
    ...  
    L1:for(i=1;i<Rows; i++) {  
        L2:for(j=0;j<Cols; j++) {  
            #pragma HLS PIPELINE  
            out[i][j] = in1[i][j] + in2[i][j];  
            L3:for(k=0;k<8;k++) {  
                ...Bit Level Operations on a Pixel...  
            }  
        }  
    }  
}
```

Slide 13-18:

Pipelining and Loop Flattening



- Pipeline will seek to auto-flatten upper Loops
 - For example, pipeline loop L2
 - The loops above L2 in this loop nest will be flattened
 - Removes loop transition cycles
 - Allows loop nest to process one sample per clock without *bubbles*

```
void foo(in1[N][M], in2[N][M], ...) {
...
    L1:for(i=1;i<N;i++) {
        L2:for(j=0;j<M; j++) {
#pragma HLS PIPELINE
            out[i][j] = in1[i][j] + in2[i][j];
        }
    }
}
```

```
void foo(in1[N][M], in2[N][M], ...) {
...
    L1_L2:for(k=1;k<N*M; k++) {
#pragma HLS PIPELINE
        out[k] = in1[k] + in2[k];
    }
}
```

- L2 is now a top-level loop and can use rewind option
- Only perfect and semi-perfect loops can be flattened

Slide 13-19:

Sample Based: Pipeline the Function



- Function
 - Function **must** be pipelined, else it cannot read one sample per clock
 - Function is called once per sample
 - Loops must be allowed to unroll
 - Can contain loops but the loops perform internal calculations; e.g. accumulations, with no I/O accesses and are generally unrolled
 - Loops with variable bounds cannot be unrolled and must be re-coded to allow unrolling (use a max limit with an "if-condition-then-break")
- For cases with hierarchy
 - Pipeline each sub-function and sub-loop
 - Then dataflow all the sub-tasks
 - Random logic will be grouped into a one-cycle task

```
void foo(in1, out1, ...) {
...
#pragma HLS PIPELINE
L1:for(i=1;i<N;i++) {
    L2:for(j=0;j<M; j++) {
        out1 = in1 + tmp[i][j];
    }
}
}
```

Slide 13-20:

Pipelining and Hierarchy: Keep in Mind!



- Pipelining unrolls loops
 - All loops below the level of the pipeline are automatically unrolled
 - Creates more operations to schedule

Pipeline Function foo

```
void foo(in1[N][M],  
        in2[N][M], ...) {  
#pragma HLS PIPELINE  
    ...  
    L1:for(i=1;i<N;i++) {  
        L2:for(j=0;j<M; j++) {  
            out[i][j] = in1[i][j]  
            + in2[i][j];  
        }  
    }  
}
```

Pipeline Loop L1

```
void foo(in1[N][M],  
        in2[N][M], ...) {  
    ...  
    L1:for(i=1;i<N;i++) {  
        #pragma HLS PIPELINE  
        L2:for(j=0;j<M; j++) {  
            out[i][j] = in1[i][j]  
            + in2[i][j];  
        }  
    }  
}
```

Pipeline Loop L2

```
void foo(in1[N][M],  
        in2[N][M], ...) {  
    ...  
    L1:for(i=1;i<N;i++) {  
        L2:for(j=0;j<M; j++) {  
            #pragma HLS PIPELINE  
            out[i][j] = in1[i][j]  
            + in2[i][j];  
        }  
    }  
}
```



Auto-Unrolls Loop 1
Auto-Unrolls Loop 2
 M^N Adder
 $3^M N$ Port Operations

Auto-Unrolls Loop 2
M Adder
 3^M Port Operations

1 Adder
3 Port Operations

Slide 13-21:

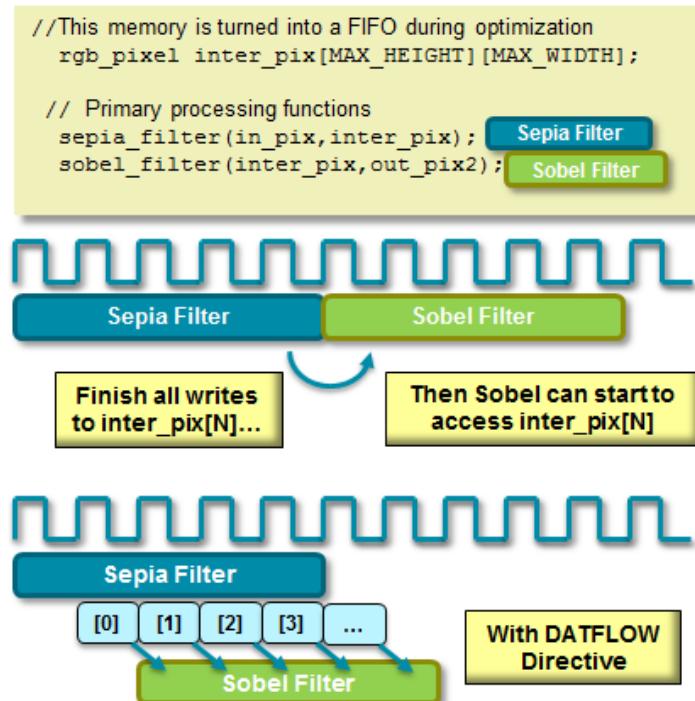
Review: Dataflow



- Default operation of synthesized design
 - Complete the execution of one function/loop
 - Then start the execution of next function/loop

- Dataflow
 - Allows the next function/loop/task to start as soon as data is ready
 - Interval is improved
 - Functions/loops will operate in parallel
 - Buffers data between processes
 - Worst case 2-BRAM (ping-pong)
 - Optimized case, 1 reg (1 element FIFO)

- Example shows functions but same principle applies to loops



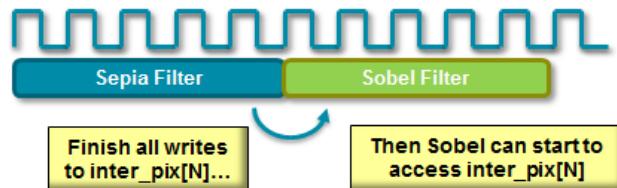
Slide 13-22:

Task Parallelism Review: Dataflow



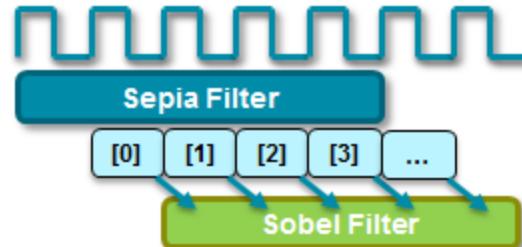
- Arrays are passed as single entities by default

```
//This memory is turned into a FIFO during
optimization
rgb_pixel inter_pix[MAX_HEIGHT] [MAX_WIDTH];
// Primary processing functions
sepia_filter(in_pix,inter_pix);
sobel_filter(inter_pix,out_pix);
```



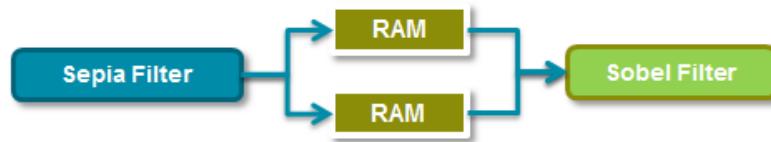
- Dataflow pipelining allows Sobel to start when data is ready

- Interval is improved
- Functions will operate in parallel



- Dataflow creates memory channels

- Created between the loops or functions to store data samples
- Default is a ping-pong buffer memory
 - Safe and reliable default
 - A FIFO can also be used



Slide 13-23:

Dataflow Limitations



- Single-point producers and consumers for arrays or streams
 - Dataflow only supports single-point connections
 - Edit code to ensure this condition satisfied



```
void foo(in[N][M], out1[N][M],  
        out2[N][M]) {  
    // Duplicated data  
    L1:for(i=1; i<N; i++) {  
        L2:for(j=0; j<M; j++) {  
            out1[i][j] = in[i][j];  
            out2[i][j] = in[i][j];  
        }  
    }  
}
```

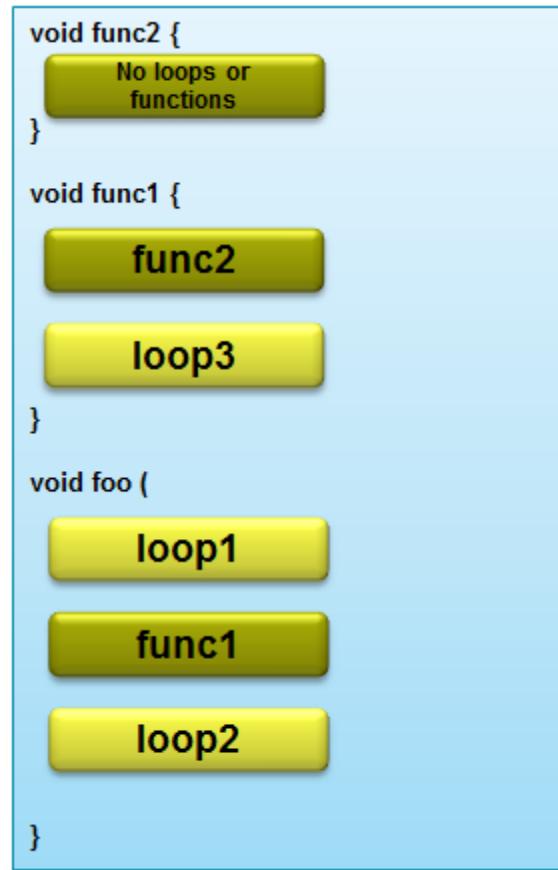
- Dataflow cannot be performed with conditionals
 - Data must flow unconditionally between tasks
- Dataflow regions
 - Tasks within functions can be optimized with dataflow
 - At any level of hierarchy
 - Tasks within loops cannot be optimized with dataflow

Slide 13-24:

Dataflow Example: Standard Hierarchy



- func2
 - No sub-loops, no question, pipeline func2
- func1
 - func2 is already pipelined
 - Pipeline Loop3
 - Dataflow func1
 - This will operate func2 and loop3 in parallel
 - Select the optimum memory channels
- Top level
 - func1 is already using dataflow
 - Pipeline loop1 and loop3
 - Dataflow function foo
 - Operate loop1, func1, and loop2 in parallel
 - Select the optimum memory channels



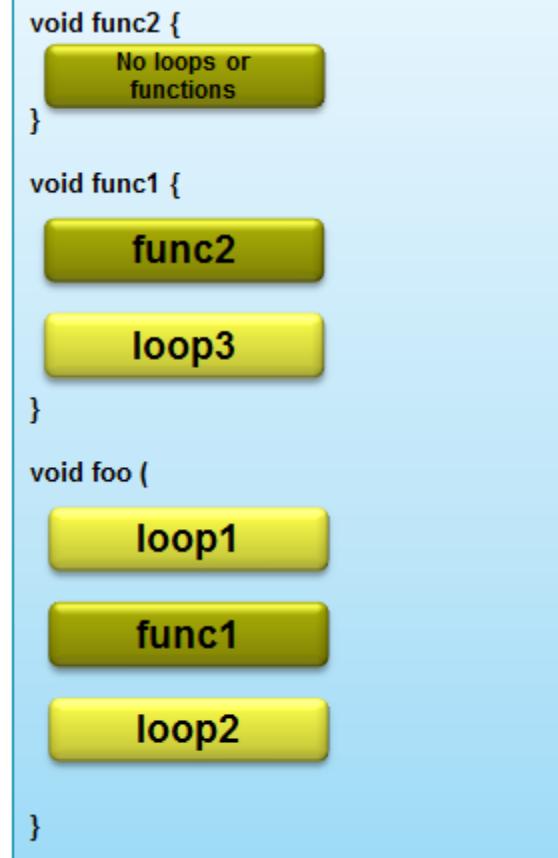
Typical Hierarchy of Functions and Loops

Slide 13-25:

Dataflow Example: Inline Example



- func2
 - No sub-loops, no question, pipeline func2
- func1
 - Inline func1
 - This raises func2 and loop3 into foo
- Top level
 - func1 is inlined
 - func2 is already pipelined
 - Pipeline loop3
 - Pipeline loop2
 - Dataflow function foo
 - Operate loop1, func2, loop3, and loop2 in parallel
 - Select the optimum memory channels



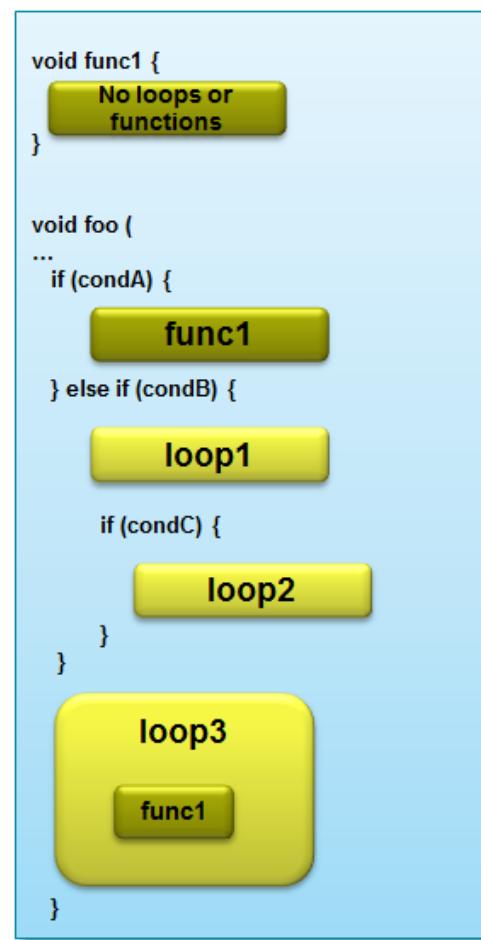
Typical Hierarchy of Functions and Loops

Slide 13-26:

Dataflow Example: Conditional Tasks



- Pipeline all tasks
 - func1, loop1, loop2, and loop3
- Dataflow function foo
- Conditionals
 - Prevent dataflow between tasks
 - All tasks in the conditional section will be grouped
 - loop2 leads to the following
 - Conditional tasks
 - Conditionally executed one after the other
- Push conditionals into tasks



Slide 13-27:

Controlling Dataflow Memory Channels



- Dataflow creates memory channels
 - Default is a ping-pong buffer memory
 - A FIFO can also be used
- Ping-pong buffer
 - Creates two block RAMs for every array being transferred
 - Can substantially increase the size of the design
- FIFO buffer
 - Use a FIFO buffer for streaming designs
 - Size of the FIFO is user specified
- Methodology for streaming high-throughput designs
 - Use ping-pong buffers by default: safe
 - Once RTL simulation passes, change to FIFOs (config_dataflow)
 - Reduce FIFO size, ensure RTL simulation does not stall
 - Else FIFO is too small

Slide 13-28:

Step 3: Optimize Structures for Performance



Directives and Configurations	Description
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers to improve access to data and remove block RAM bottlenecks.
DEPENDENCE	Used to provide additional information that can overcome loop-carry dependencies and allow loops to be pipelined (or pipelined with lower intervals).
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.
UNROLL	Unroll <i>for</i> loops to create multiple independent operations rather than a single collection of operations.

Slide 13-29:

Step 3: Optimize Structures for Performance (2)



Directives and Configurations	Description
Config Array Partition	Determines how arrays are partitioned, including global arrays and if the partitioning impacts array ports.
Config Compile	Controls synthesis-specific optimizations such as the automatic loop pipelining and floating point math optimizations.
Config Schedule	Determines the effort level to use during the synthesis scheduling phase and the verbosity of the output messages.

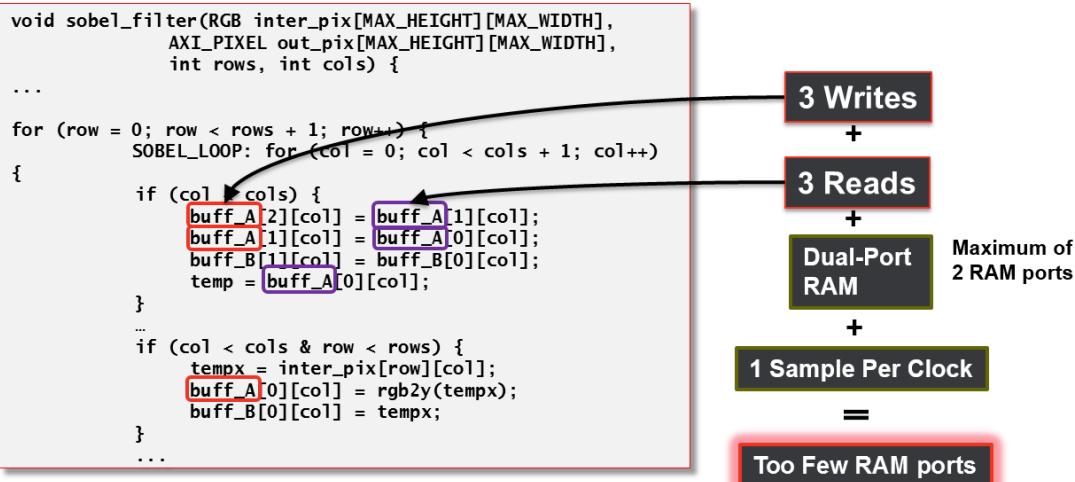
Slide 13-30:

Bandwidth Issues



- Array accesses (block RAM) can be bottlenecks inside functions or loops
 - Still prevents a II of 1 despite PIPELINE and DATAFLOW
 - Prevents processing of one sample per clock

```
@I [SCHED-61] Pipelining loop 'SOBEL_LOOP'.
@W [SCHED-69] Unable to schedule 'store' operation (image_demo.cpp:172) of
variable 'y' on array 'buff_A' due to limited resources (II = 2).
@I [SCHED-61] Pipelining result: Target II: 1, Final II: 3, Depth: 13.
```



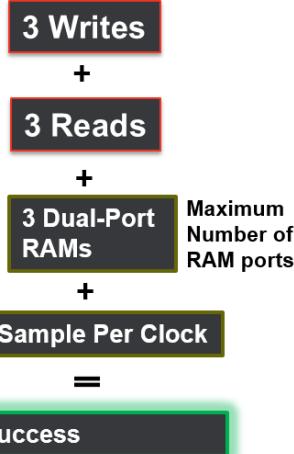
Slide 13-31:

Solution: Partitioning Arrays



- RAMs can create bottlenecks inside a function or loop
 - Partitioning improves availability of data
 - Similar for a sub-function, it may have too few ports

```
void sobel_filter(RGB inter_pix[MAX_HEIGHT][MAX_WIDTH],
                  AXI_PIXEL out_pix[MAX_HEIGHT][MAX_WIDTH],
                  int rows, int cols) {
    ...
    #pragma AP_ARRAY_PARTITION variable=buf_A dim=1 complete
    for (row = 0; row < rows + 1; row++) {
        SOBEL_LOOP: for (col = 0; col < cols + 1; col++) {
            if (col < cols) {
                buff_A[2][col] = buff_A[1][col];
                buff_A[1][col] = buff_A[0][col];
                buff_B[1][col] = buff_B[0][col];
                temp = buff_A[0][col];
            }
            ...
            if (col < cols & row < rows) {
                tempx = inter_pix[row][col];
                buff_A[0][col] = rgb2y(tempx);
                buff_B[0][col] = tempx;
            }
            ...
        }
    }
}
```



Slide 13-32:

Data Dependencies



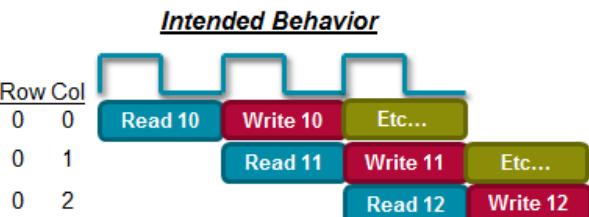
- Design works fine if not pipelined

```

for (row = 0; row < rows + 1; row++) {
    L2: for (col = 0; col < cols + 1; col++) {
        if (col < cols) {
            Line: 162 buff_A[2][col] = buff_A[1][col];
            Line: 163 buff_A[1][col] = buff_A[0][col];
            buff_B[1][col] = buff_B[0][col];
            temp = buff_A[0][col];
        }
    }
}

```

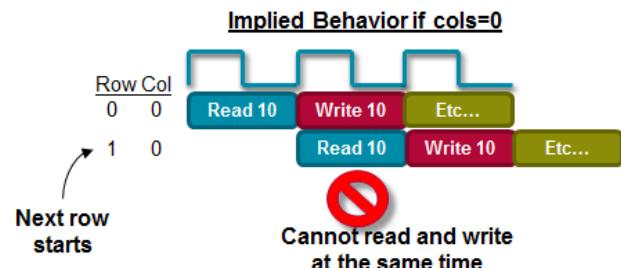
Buff A[1][col]



- Pipelined: implied dependency

```
@W [SCHED-68] Unable to enforce a carried dependency constraint between 'store' operation ('image_demo.cpp:163') and 'load' operation ('buff_A_1_load', 'image_demo.cpp:162')
```

- Index limit "cols" is an input
- What if cols=0, it means...
 - There is only one iteration of loop L2



Slide 13-33:

Code Re-Write to Improve Bandwidth



- Intuitive code
 - This example algorithm works on a window of data
 - Requires a sub-loop inside the sample loop to fill the window
 - Input is read 27X per pixel
 - Port is accessed 27X per pixel

```
void image(in_pix[N][M], ...) {  
    ...  
    for(row = 0; row < MAX_HEIGHT+1; row++){  
        for(col = 0; col < MAX_WIDTH+1; col++){  
            #pragma HLS PIPELINE  
            // Create Sobel Window  
            y_win_fill: for(i = -1; i < 2; i++){  
                x_win_fill: for(j = -1; j < 2; j++){  
                    pix.R = in_pix[(row+i)][(col+j)].R;  
                    pix.G = in_pix[(row+i)][(col+j)].G;  
                    pix.B = in_pix[(row+i)][(col+j)].B;  
                    ...  
                }  
            }  
        }  
    }  
}
```

- Optimized code
 - A line buffer is created
 - Input is read once per pixel
 - The line buffer logic ensures the window is populated and managed

```
void image(in_pix[N][M], ...) {  
    ...  
    for(row = 0; row < MAX_HEIGHT+1; row++){  
        for(col = 0; col < MAX_WIDTH+1; col++){  
            #pragma HLS PIPELINE  
            // Create Sobel Window  
            //Line Buffer fill  
            if(col < MAX_WIDTH){  
                line_buffer[2][col] = line_buffer[1][col];  
                line_buffer[1][col] = line_buffer[0][col];  
            }  
            if(col < MAX_WIDTH & row < MAX_HEIGHT){  
                line_buffer[0][col] = int_pix[row][col];  
            }  
        }  
    }  
}
```

Slide 13-34:

Specifying Dependencies



- The column input size will never be zero
 - Vivado HLS tool cannot know that or determine by analysis
 - Issue is only highlighted when the code is pipelined
 - There is no issue unless the design is pipelined
- Vivado HLS tool allows implied dependencies to be marked false
 - Allows designers to inform tool of *external* conditions
 - In this case, the input column size will never be zero
 - There are no dependencies **between** loop iterations

```
for (row = 0; row < rows + 1; row++) {
    SOBEL_LOOP: for (col = 0; col < cols + 1; col++) {

        // No Dependence Between Loop Iterations
        #pragma HLS dependence variable=buf_A inter=false
        #pragma HLS dependence variable=buf_B inter=false

    }
}
```

- Dependencies also can be within a loop iteration

Slide 13-35:

Step 4: Reduce Latency



Directives and Configurations	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.



When the Vivado HLS tool finishes minimizing the initiation interval, it automatically seeks to minimize the latency. The optimization directives listed in the table above can help reduce or specify a particular latency.

These are generally not required when the loops and function are *pipelined* as in most applications the latency is not critical; throughput typically is.

If the loops and functions are not pipelined, the throughput will be limited by the latency, because the task will not start reading the next set of inputs until the prior task has completed.

The LATENCY directive is used to specify the required latency. The loop optimization directives can be used to flatten a loop hierarchy or merge serial loops together. The benefit to the latency is due to the fact that it typically costs a clock cycle to enter and leave a loop. The fewer the number of transitions between loops, the fewer number of clock cycles a design will take to complete.

Slide 13-36:

Step 5: Improve Area



Directives and Configurations	Description
ALLOCATION	Specify a limit for the number of operations, cores or functions used. This can force the sharing of hardware resources and might increase latency.
ARRAY_MAP	Combines multiple smaller arrays into a single large array to help reduce block RAM resources.
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.
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing and improve logic optimization.
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.
RESOURCE	Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL.



The ALLOCATION and RESOURCE directives are used to limit the number of operations and to select which cores (or resources) are used to implement the operations. For example, you could limit the function or loop to using only one multiplier and specify it to be implemented using a pipelined multiplier. The binding configuration is used to globally limit the use of a particular operation.

If the ARRAY_PARTITION directive is used to improve the initiation interval, you might want to consider using the ARRAY_RESHAPE directive instead. The ARRAY_RESHAPE optimization performs a similar task to array partitioning; however the reshape optimization recombines the elements created by partitioning into a single block RAM with wider data ports.

If the C code contains a series of loops with similar indexing, merging the loops with the LOOP_MERGE directive might allow some optimizations to occur.

Finally, in cases where a section of code in a pipeline region is only required to operate at an initiation interval lower than the rest of the region, the OCCURRENCE directive is used to indicate this logic can be optimized to execute at a lower rate.

Slide 13-37

Step 5: Improve Area (2)



Directives and Configurations	Description
STREAM	Specifies that a specific memory channel is to be implemented as a FIFO or RAM during dataflow optimization.
Config Bind	Determines the effort level to use during the synthesis binding phase and can be used to globally minimize the number of operations used.
Config Dataflow	This configuration specifies the default memory channel and FIFO depth in dataflow optimization.

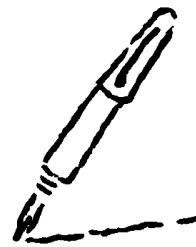
Slide 13-38:

Summary



- HLS UltraFast Design Methodology
 - Collection of good practices
 - Achieve results quickly
 - Accelerate time to mark
 - Step 1: Add initial optimization directives
 - Step 2: Pipeline for performance
 - Step 3: Optimize structures for performance
 - Step 4: Reduce latency
 - Step 5: Improve area

Capture Your Notes Here



Appendix: SystemC Synthesis

Slide 14-1:

2016.1

This module is an introduction to the SystemC synthesis support in the Vivado® HLS tool. It also provides some coding style guidelines for working with SystemC.

Slide 14-2:

SystemC Synthesis Coverage



- **SystemC Synthesis Coverage**
 - SystemC Constructs
 - TLM Synthesis
 - Multi-Clock Domains
 - Vivado HLS Tool SystemC Extensions
 - Tutorial Example
 - Coding Styles for SystemC (Complements to C Coding Style)
 - Differences Between Vivado HLS Tool and SystemC Types

Slide 14-3:

SystemC Synthesis Coverage



- C/C++
 - All synthesizable C/C++ constructs are supported in the SystemC flow
 - Unique advantage of unified C/C++/SystemC support in the Vivado HLS tool

- SystemC
 - SystemC support in the Vivado HLS tool is based on
 - IEEE1666-2005
 - OSCI SystemC 2.2 library
 - OSCI SystemC synthesizable subset 1.3 (draft)

Slide 14-4:

SystemC Synthesis Coverage



- Data type and operations

Category	Constructs
C/C++ data types	<ul style="list-style-type: none"> • bool, (unsigned)char, (unsigned)short, (unsigned)int, (unsigned)long, (unsigned)long long, float, double, etc.
Integer data types	<ul style="list-style-type: none"> • sc_int, sc_uint, • sc_bignum, sc_bignum (up to 2^{23} bits)
Fixed point data types	<ul style="list-style-type: none"> • sc_fixed, sc_ufixed
Logical data types	<ul style="list-style-type: none"> • sc_bit, sc_bv, sc_logic, sc_lv
Constants	<ul style="list-style-type: none"> • 1234516121311140ULL; Big constant integer can be read from string
Arithmetic and logical operators on sc_data types	<ul style="list-style-type: none"> • +, -, *, /, %, +=, -=, *=, /=, %=, >, <, >>, <<, !=, ==, &&, , &, &=, !==
Operation methods	<ul style="list-style-type: none"> • To integer: to_int(), to_uint(), to_int64(), to_uint64() • Partselect: range(hi, lo); Bit select: x[i]; Concatenation: (x, y), concat(x, y) • Reduce: and_reduce(), nand_reduce(), or_reduce() nor_reduce(), xor_reduce() • Reverse: reverse() • Rotate: rrotate(), lrotate()

Slide 14-5:

SystemC Synthesis Coverage



- SystemC constructs

Category	Constructs
Modules	• class <code>sc_module</code> , or macro <code>SC_MODULE</code> , which specifies a hardware module
Channels	• <code>sc_signal<T></code> , <code>sc_fifo<T></code> , <code>tlm_fifo<T></code> , <code>sc_buffer<T></code>
Ports and interfaces	• <code>sc_in<T></code> , <code>sc_out<T></code> , <code>sc_inout<T></code> , <code>sc_fifo_in/out<T></code>
Processes	• <code>SC_METHOD</code> , <code>SC_CTHREAD</code>
Constructors	• <code>SC_CTOR</code> , constructor
Wait statements	• <code>wait()</code> (in <code>SC_CTHREAD</code>)
Module instantiations	• Statically allocated members defined in the current module declaration
Sensitivity	• <code>sc_sensitive</code> , <code>sc_sensitive_neg</code> , <code>sc_sensitive_pos</code>
TLM interfaces	• OSCITLM interfaces: <code>tlm_fifo_put_if</code> , <code>tlm_fifo_get_if</code> , <code>tlm_blocking_get_if</code> , <code>tlm_blocking_put_if</code> • AutoESL extended interfaces: memory interface and bus interface • Composite port defined TLM interfaces

Slide 14-6:

SystemC Constructs



- SystemC Synthesis Coverage
- **SystemC Constructs**
- TLM Synthesis
- Multi-Clock Domains
- Vivado HLS Tool SystemC Extensions
- Tutorial Example
- Coding Styles for SystemC (Complements to C Coding Style)
- Differences Between Vivado HLS Tool and SystemC Types

Slide 14-7:

SystemC Constructs



- SC_MODULE
 - SystemC module represents an individual identifiable hardware element
 - Template SC_MODULE can only be sub modules
- SC_CTOR
 - SC_CTOR is a macro used to declare module constructor
 - Module constructor can also be declared with SC_HAS_PROCESS macro
 - Module's constructor cannot contain any functionality
 - Only used to declare processes and module instantiations

Slide 14-8:

SystemC Constructs



- Clock and reset
 - Having all SC_MODULE constructs with explicit clock and reset port is recommended
 - Vivado HLS tool can insert reset signal automatically, but not recommended
 - All clock/reset ports or signals must be driven by top-level ports

```
SC_MODULE(mod){  
    sc_in<bool> clk;  
    sc_in<bool> reset;  
  
    void dummy();  
    SC_CTOR(mod){  
        SC_CTHREAD(dummy, clk.pos());  
        reset_signal_is(reset, true);  
    }  
};
```

Slide 14-9:

SystemC Constructs



■ Ports

- Ports represent the externally visible interface to a module and are used to transfer data into and out of the module. Ports can be declared using `sc_in`, `sc_out`, and `sc_inout` constructs
- Vivado HLS tool provides extended memory interface, bus interfaces, and related simulation libraries

```
SC_MODULE(mod){  
    sc_in<bool> clk;  
    sc_in<bool> reset;  
    sc_fifo_in <sc_uint<8>> din;  
    sc_out<sc_uint<8>> dout;  
    sc_out<bool> dout_rdy;  
  
    void dummy();  
};  
SC_CTOR(mod){  
    SC_CTHREAD(dummy, clk.pos());  
    reset_signal_is(reset, true);  
};
```

Slide 14-10:

SystemC Constructs



- Processes
 - SC_METHOD
 - Treated exactly as its RTL semantic. That is, all of the operations with one SC_METHOD will be executed when the process is triggered
 - Vivado HLS tool can automatically determine whether a SC_METHOD is combinational or sequential, according to its sensitivity specification, and generate the correct sequential behavior

```
SC_METHOD(process);  
sensitive << clk.pos() << reset.pos();
```

```
Void process() // SC_METHOD process  
{  
    if (reset.read() == 0) {  
        <reset block>  
    }  
    else {  
        <process functionality>  
    }  
}
```

Slide 14-11:

SystemC Constructs



- Processes
 - Vivado HLS tool supports SC_CTHREAD and SC_METHOD for specifying concurrent processes (see OSCI synthesis subset 1.3)
 - SC_CTHREAD
 - SC_CTHREAD is mainly for specifying a multi-cycle process, triggered on one edge of a single clock
 - Within an SC_CTHREAD, can specify protocol regions for timed I/O protocols by using wait() statement, and the synthesis engine will honor the specified timing

```
void process() //SC_CTHREAD process
{
    <reset behavior>
    wait();
    <post reset initializations(optional)>

    while(1) // the main loop
    {
        wait();
        <process functionality>
    }
}
```

Slide 14-12:

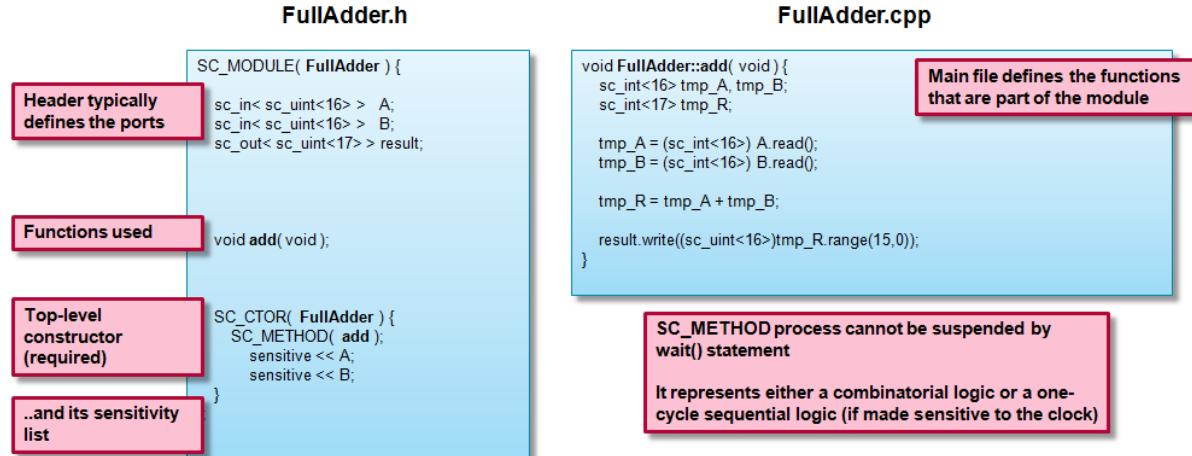
SystemC Constructs



- Channels
 - sc_signal / sc_buffer
 - Supported for multi-thread communications
 - Standard variables should not be used for communication between thread due to delta time concept—use sc_signal or sc_buffer
 - sc_fifo / tlm_fifo
 - Support provided for methods
 - Non-blocking read/write
 - Blocking read/write
 - num_available()/num_free()
 - nb_can_put()/nb_can_get()
 - All channel arrays should be explicitly partitioned or synthesis will fail

Slide 14-13:

FullAdder Example Behavioral Style



Slide 14-14:

FullAdder Example Structural Hierarchy



The structural model
“Instantiates” the modules

Functions used

Connectivity

...and it's sensitivity list

FullAdder.h

```
SC_MODULE(full_adder) {
    sc_in<bool> a, b, carry_in;
    sc_out<bool> sum, carry_out;
    sc_signal<bool> c1, s1, c2;

    half_adder ha1, ha2;

    void prc_or();
};

SC_CTOR(full_adder) :
    ha1("ha1"),
    ha2("ha2") {

    ha1.a(a);
    ha1.b(b);
    ha1.sum(s1);
    ha1.carry(c1);

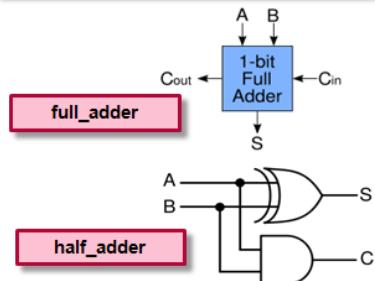
    ha2 << s1 << carry_in << sum << c2;

    SC_METHOD(prc_or);
    sensitive << c1 << c2;
};
```

FullAdder.cpp

```
void full_adder::prc_or() {
    carry_out = c1 | c2;
}

void half_adder::prc_half_adder() {
    bool s, c;
    s=a.read() ^ b.read();
    c=a.read() & b.read();
    sum.write(s);
    carry.write(c);
}
```



Slide 14-15:

TLM Synthesis



- SystemC Synthesis Coverage
- SystemC Constructs
- **TLM Synthesis**
- Multi-Clock Domains
- Vivado HLS Tool SystemC Extensions
- Tutorial Example
- Coding Styles for SystemC (Complements to C Coding Style)
- Differences Between Vivado HLS Tool and SystemC Types

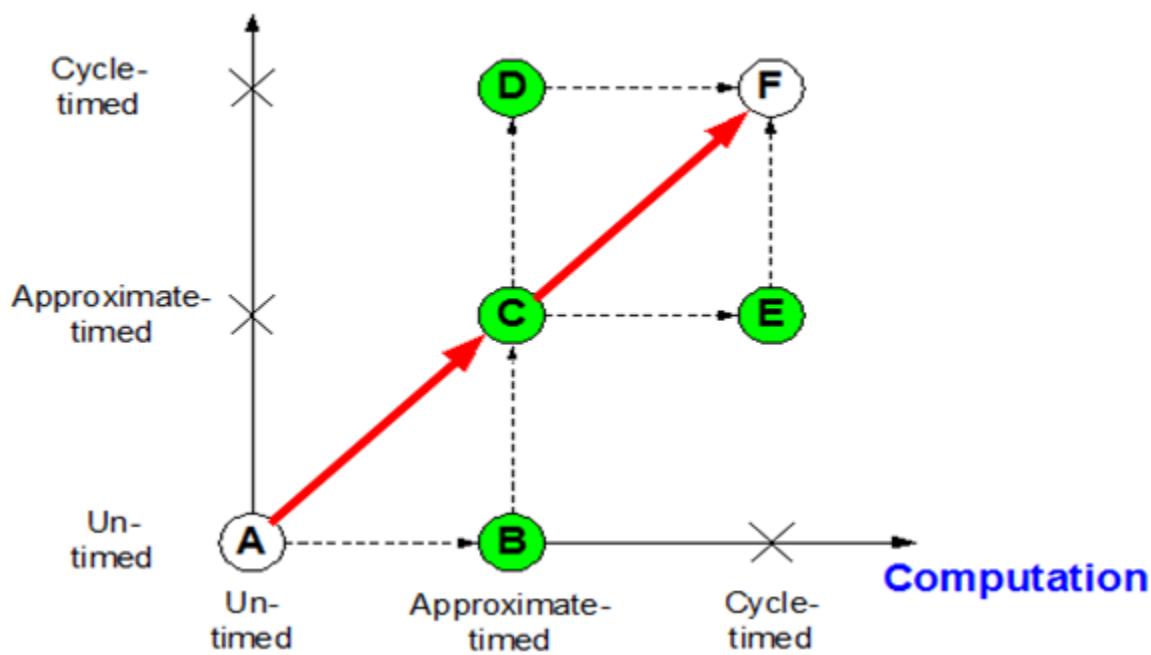
Slide 14-16:

Abstraction Levels



- SystemC modeling abstractions
 - SystemC and OSCI TLM standards provide a wide range of modeling abstraction levels, supporting both hardware and software modeling

Communication



Slide 14-17:

High-Level Synthesis Abstractions



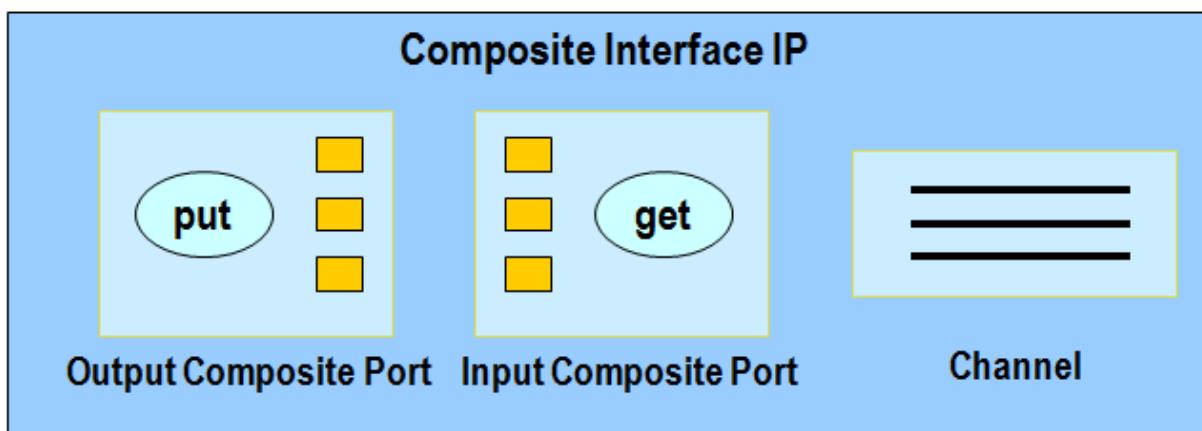
- TLM-level synthesis
 - At this level, computation algorithm can be purely untimed, while the communication between processes and modules must be modeled explicitly and communication protocol must be defined
- Signal-level synthesis
 - Signal-level model's interface is pin accurate, the same as final RTL. At this level, user needs to model the signal behavior protocols, and the accuracy of protocol signals can be verified
- RT-level synthesis
 - SystemC synthesis in the Vivado HLS tool allows RT-Level modeling, which is powerful for control-specific designs
 - Vivado HLS tool outputs an RTL design (Verilog, VHDL, and SystemC)

Slide 14-18:

TLM Synthesis Support: Composite Port (1)



- TLM synthesis with composite port
 - Packaged class that contains a set of signal-level ports (or other composite ports) as class members
- Composite interface usually consists of an input composite port, an output composite port, and a channel



Slide 14-19:

TLM Synthesis Support: Composite Port (2)



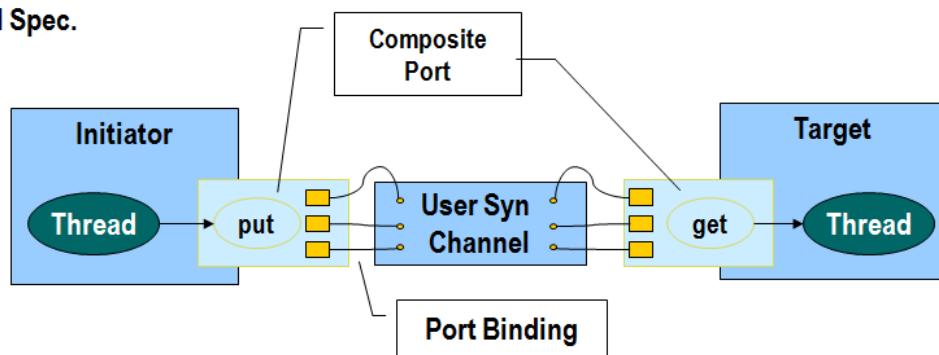
- TLM synthesis with composite port

Original TLM Specification



Switching the Channel Spec.

Synthesizable TLM Using Composite Ports



Slide 14-20:

TLM Synthesis Support

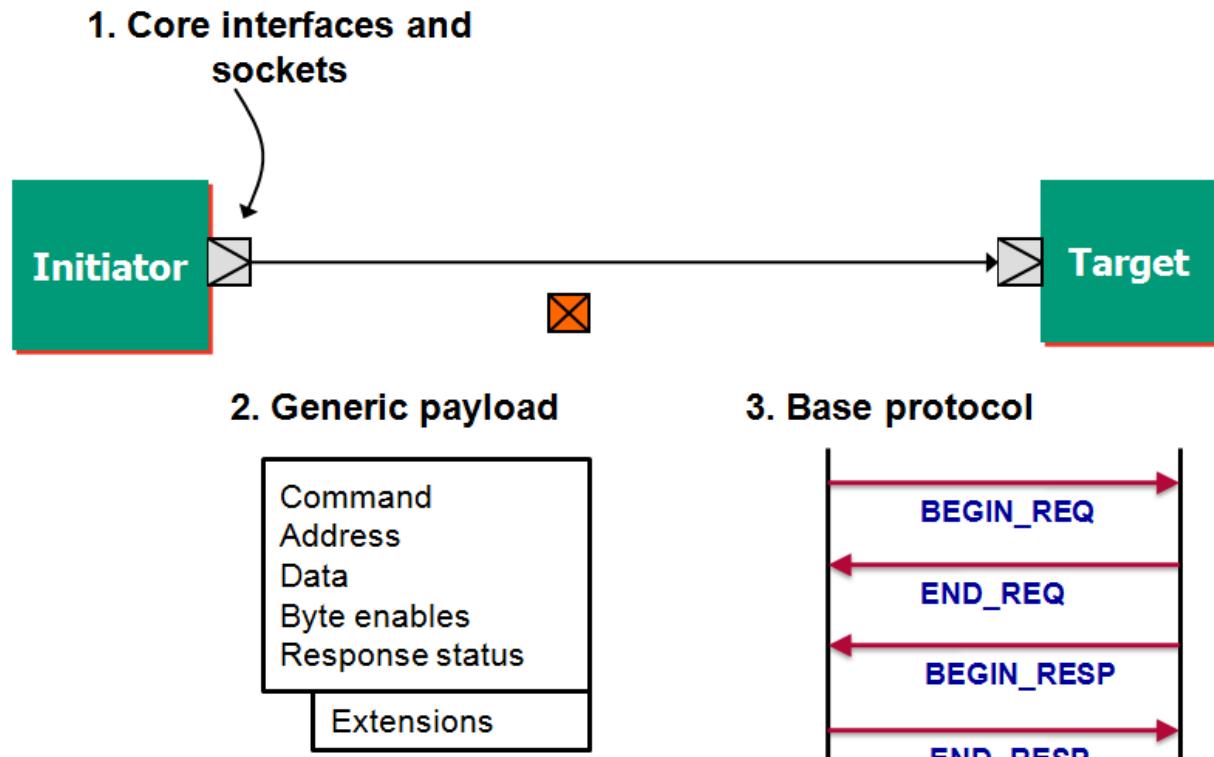


- Benefits

- With the Vivado HLS tool composite port, designers can refine their TLM interface from a simulation model to a synthesizable model without touching source code within module processes
- Vivado HLS tool provides intrinsic support for tlm_fifo, bus, memory interfaces, etc.

Slide 14-21:

TLM 2.0 Components



Maximal interoperability for memory-mapped bus models

Slide 14-22:



TLM 2.0 Wrapper Support

- TLM 2.0 wrapper support
 - Synthesis
 - Wrapper TLM 2.0 design with I/O transactors
 - Based on composite port
 - Simulation
 - Integrate to original TLM 2.0 platform with adapter

Slide 14-23:

Multi-Clock Domains

- SystemC Synthesis Coverage
- SystemC Constructs
- TLM Synthesis
- **Multi-Clock Domains**
- Vivado HLS Tool SystemC Extensions
- Tutorial Example
- Coding Styles for SystemC (Complements to C Coding Style)
- Differences Between Vivado HLS Tool and SystemC Types

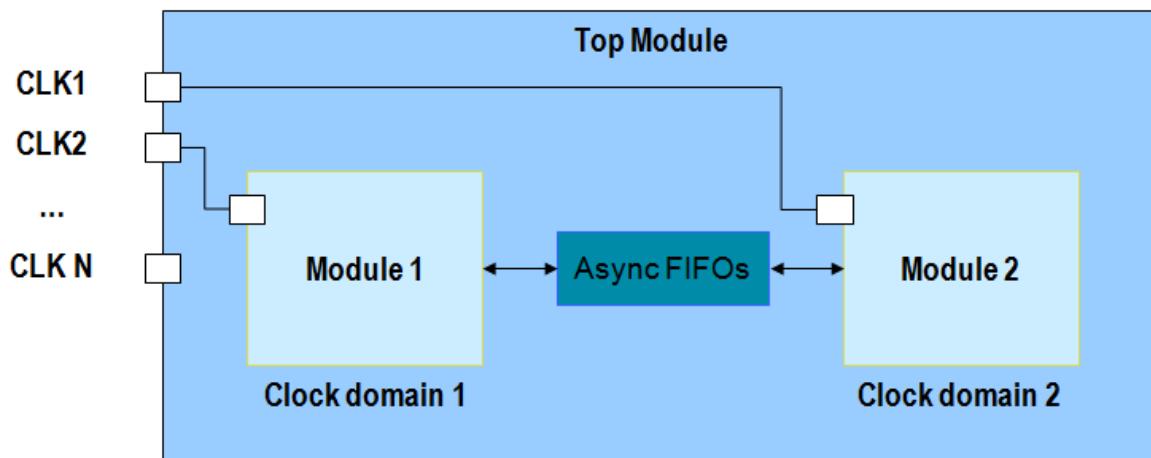


Slide 14-24:

Multi-Clock Domain Support (1)



- Multi-clock domain synthesis
 - Vivado HLS tool SystemC supports multi-clock domain across modules
 - Connect different clock domains with Async FIFOs automatically
 - Signals are also supported with synchronization in SC_METHODS



Slide 14-25:

Multi-Clock Domain Support (2)



```
SC_MODULE(IDCT_top){  
  
public:  
    // Signal ports.  
    sc_in_clk clk1;  
    sc_in_clk clk2;  
    sc_in_clk clk3;  
    sc_in<bool> rst;  
  
    // FIFO ports.  
    sc_fifo_in<sc_int<12>> DataIn;  
    sc_fifo_out<sc_int<8>> DataOut;  
  
    // Channel instances.  
    sc_fifo<sc_int<16>> Chn0, Chn1;  
  
    // Module instances.  
    IDCT_row subModule_row;  
    IDCT_col subModule_col;  
    IDCT_buffer subModule_buffer;  
    ...  
}
```

```
create_clock -period 8ns -name clk1  
create_clock -period 5ns -name clk2  
create_clock -period 12ns -name clk3  
  
set_directive_clock IDCT_row clk1  
set_directive_clock IDCT_col clk2  
set_directive_clock IDCT_buffer clk3
```

Slide 14-26:

Vivado HLS Tool SystemC Extensions

- SystemC Synthesis Coverage
- SystemC Constructs
- TLM Synthesis
- Multi-Clock Domains
- **Vivado HLS Tool SystemC Extensions**
- Tutorial Example
- Coding Styles for SystemC (Complements to C Coding Style)
- Differences Between Vivado HLS Tool and SystemC Types



Slide 14-27:

Vivado HLS Tool SystemC Extensions



- External memory port constructs
 - OSCI SystemC library has no memory interfaces

Format: ap_mem_port<data_type, addr_type, mem_size, mem_type> variable_name

```
#include "ap_mem_if.h"

SC_MODULE(dut) {
    //External Memory Port
    ap_mem_port<sc_uint<32>, sc_uint<8>, 256, RAM2P> mem_if;

    void thread()
    {
        //reset region
        wait();

        //Initialization
        int tmp;
        while(1)
        {
            do {wait();} while(!start.read());

            tmp = mem_if[addr++];
            mem_if[addr] = tmp;

            dout.write(tmp);
        }
    };
}
```

Slide 14-28:

Tutorial Example

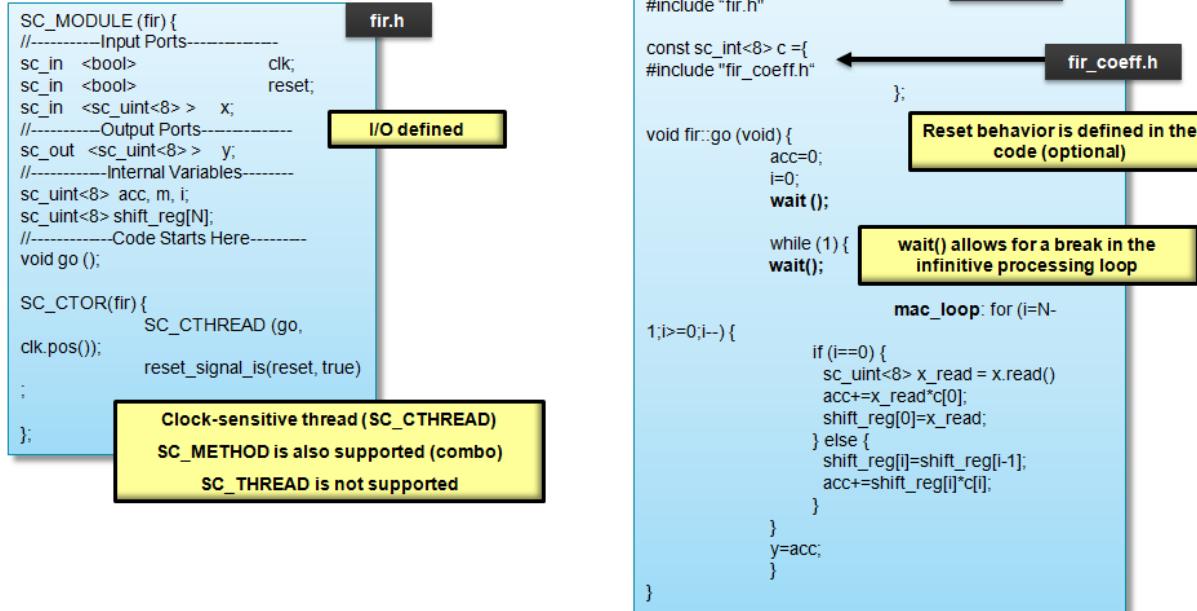
- SystemC Synthesis Coverage
- SystemC Constructs
- TLM Synthesis
- Multi-Clock Domains
- Vivado HLS Tool SystemC Extensions
- **Tutorial Example**
- Coding Styles for SystemC (Complements to C Coding Style)
- Differences Between Vivado HLS Tool and SystemC Types



Slide 14-29:

FIR Filter Example (Demo)

- Common input and output



Slide 14-30:

FIR Filter Example (Demo)

- FIFO input and output



```
SC_MODULE(fir) {
    //-----Input Ports-----
    sc_in <bool> clk;
    sc_in <bool> reset;
    sc_fifo_in <sc_uint<8>> x;
    //-----Output Ports-----
    sc_fifo_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)
        };
    }
}
```

I/O defined

**Clock-sensitive thread (SC_CTHREAD)
SC_METHOD is also supported (combo)
SC_THREAD is not supported**

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

void fir::go (void) {
    acc=0;
    i=0;
    wait();
}

while (1) {
    wait();
}
```

Reset behavior is defined in the code (optional)

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

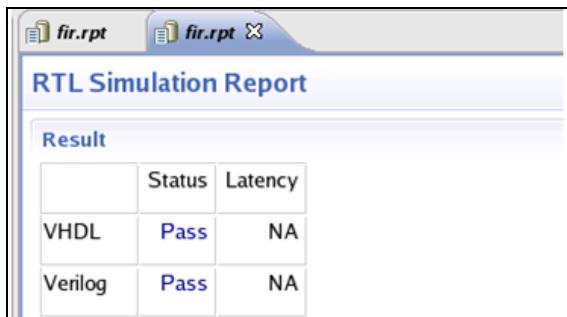
```
1;i>=0;i--) {
    mac_loop: for (i=N-1;i>=0;i--) {
        if (i==0) {
            sc_uint<8> x_read = x.read();
            acc+=x_read*c[0];
            shift_reg[0]=x_read;
        } else {
            shift_reg[i]=shift_reg[i-1];
            acc+=shift_reg[i]*c[i];
        }
    }
    y.write(acc);
}
```

Slide 14-31:

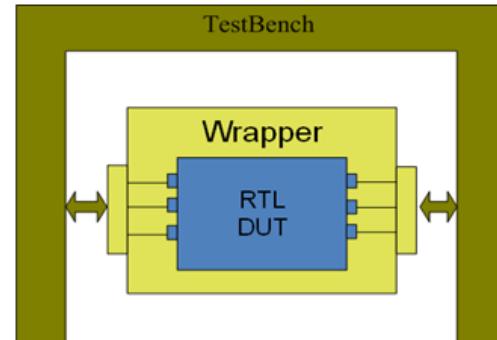
Simulation and Verification



- Run RTL simulation
 - Re-use the original ESL testbenches automatically
 - Vivado HLS tool generates a wrapper for RTL DUT automatically
 - Run Autosim to simulate and verify RTL



```
#ifdef __RTL_SIMULATION__
# include "fir_rtl_wrap.h"
# define fir_RTL_transactor
#else
#include "fir.h"
#endif
```



Slide 14-32:

Coding Styles for SystemC (Complements to C Coding Style)

- SystemC Synthesis Coverage
- SystemC Constructs
- TLM Synthesis
- Multi-Clock Domains
- Vivado HLS Tool SystemC Extensions
- Tutorial Example
- **Coding Styles for SystemC (Complements to C Coding Style)**
- Differences Between Vivado HLS Tool and SystemC Types



Slide 14-33:

Recommended Coding Styles

Complement to C Coding Style



- SC_MODULE

- It is recommended that every SC_MODULE to have clock and reset ports
 - If there is no clock or reset, Vivado HLS tool may insert clock and reset ports in RTL

```
SC_MODULE (DUT)
{
    sc_in <bool>          clock;
    sc_in <bool>          reset;

    void process1();
    void process2();

    SC_CTOR(fir)
    {
        SC_CTHREAD (process1, clock.pos());
        reset_signal_is(reset, true);

        SC_METHOD(process2);
        sensitive<<clock.pos()<<reset.pos();
    }
};
```

Slide 14-34:

Recommended Coding Styles

Complement to C Coding Style



■ SC_CTHREAD

- Put one *wait* after reset behavior region
- Main loop requires a *wait()* by OSCI lib
 - Putting it at the beginning of the main loop is recommended, instead of the end of the main loop

```
void process() //SC_CTHREADprocess
{
    <reset behavior>
    wait();           // put a wait() after reset region
    <post reset initializations(optional)>

    while(1) // the main loop
    {
        wait();           // put the required 'wait()' at the beginning of the main loop,
                           // instead of the end of the main loop
        <process functionality>
    }
}
```

Slide 14-35:

Recommended Coding Styles

Complement to C Coding Style



■ Wait for a signal

- Use *do-wait-while* statement, which can be pipelined
- *while-wait* statement is not recommended

```
void process() //SC_CTHREADprocess
{
    while(1) // the main loop
    {
        ...
        do {wait();} while(!din_vld); //use 'do-wait-while', instead of 'while-wait'
    }
}
```

Slide 14-36:

Recommended Coding Styles

Complement to C Coding Style



- Inter-process communication via share signals, not share variables
 - Use sc_signal for communications
 - Native variables can cause nondeterministic simulation behavior; they are deprecated for inter-process communications

```
SC_MODULE(test)
{
    //Ports
    sc_in<bool> clock; //clock input
    sc_in<bool> reset;
    ...

    //Variables
    sc_signal<sc_uint<3>> cnt;
    sc_signal<bool> cnt_start;

    ...
};
```

Slide 14-37:

Recommended Coding Styles

Complement to C Coding Style



- Multiple SC_METHODs cannot write to one share signal/variable
 - Multiple writer will cause multi-driver problem
 - Same as synthesizable RTL coding style

```
SC_MODULE(test)
{
    //Variables
    sc_signal<sc_uint<3>> cnt;

    void method1()
    {
        int x;
        ...
        cnt = x;      //the first driver for 'cnt'
    }

    void method2()
    {
        int y;
        ...
        cnt = y;      //the second driver for 'cnt'
    }
};
```

Slide 14-38:

Recommended Coding Styles

Complement to C Coding Style



- Do not read to output port for synthesis
 - Reading to output port will get two ports in RTL
 - Due to a limitation of VHDL, Vivado HLS tool generates two ports: one for output, one for input

```
SC_MODULE(DUT)
{
    sc_in<T> in0;
    sc_out<T> out0;
    ...
    void process()
    {
        int var=in0.read()+out0.read(); // RTL will have 2 ports (out0_i & out0_o)
        out0.write(var);
    }
};
```

Slide 14-39:

Recommended Coding Styles

Complement to C Coding Style

- Include DUT in testbench as below
 - For RTL simulation, testbench will include the RTL wrapper



```
#include <systemc.h>

#include "AESL_clock.h"
#include "driver.h"

#ifndef __RTL_SIMULATION__
#include "dut_rtl_wrap.h"
#define dut dut_RTL_transactor
#else
#include "dut.h"
#endif

SC_MODULE(System)
{
    ...
    dut U_top;
    ...
};
```

Slide 14-40:

Differences Between Vivado HLS Tool and SystemC Types

- SystemC Synthesis Coverage
 - SystemC Constructs
 - TLM Synthesis
 - Multi-Clock Domains
 - Vivado HLS Tool SystemC Extensions
 - Tutorial Example
 - Coding Styles for SystemC (Complements to C Coding Style)
 - **Differences Between Vivado HLS Tool and SystemC Types**
- 

Slide 14-41:

Differences Between Arbitrary Precision Types



- Arbitrary precision types
 - Vivado HLS tool provides arbitrary precision types for C and C++
 - SystemC provides arbitrary precision types
- Differences
 - Differences in how Vivado® HLS tool and SystemC types are implemented
- Design migration
 - Users can migrate designs from one type of C to another
 - SystemC types can be used directly in C++ designs
 - Most users will write with one set of types **or** the other
 - The following is provided as reference material for anyone who wishes to migrate code between C++ and SystemC (or vice versa)

Slide 14-42:

Summary of Arbitrary Precision Type Differences



- Differences with Vivado HLS tool and SystemC data types
 - Default constructor
 - Integer division
 - Integer modulus
 - Negative shifts
 - Over-left shift
 - Range operation
 - Fixed-point division
 - Fixed-point shifting

Slide 14-43:

Default Constructors



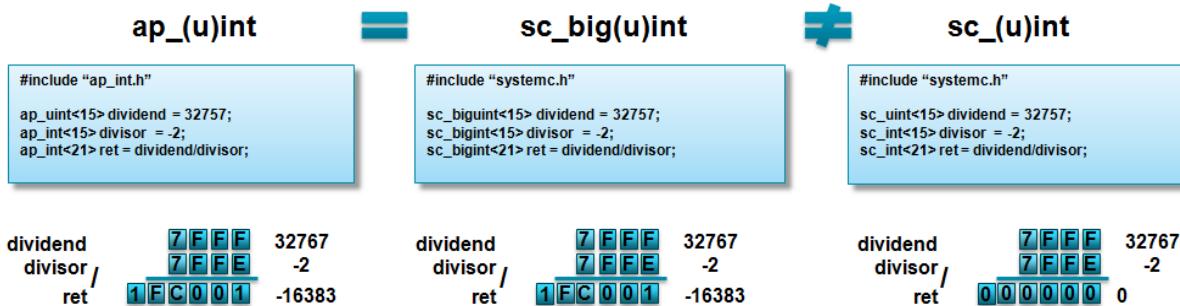
- Vivado HLS tool bit-accurate data types
 - ap_(u)int – no default initialization
 - ap_(u)fixed – no default initialization
- SystemC bit-accurate data types
 - sc_(u)int – default initialization to 0
 - sc_(u)fixed – default initialization to 0

Slide 14-44:

Division Operator



- Vivado HLS tool is consistent with sc_biguint
 - sc_big(u)int can have an arbitrary bit length
- Both behave differently from sc_(u)int
 - sc_(u)int can only have a maximum of 64 bits

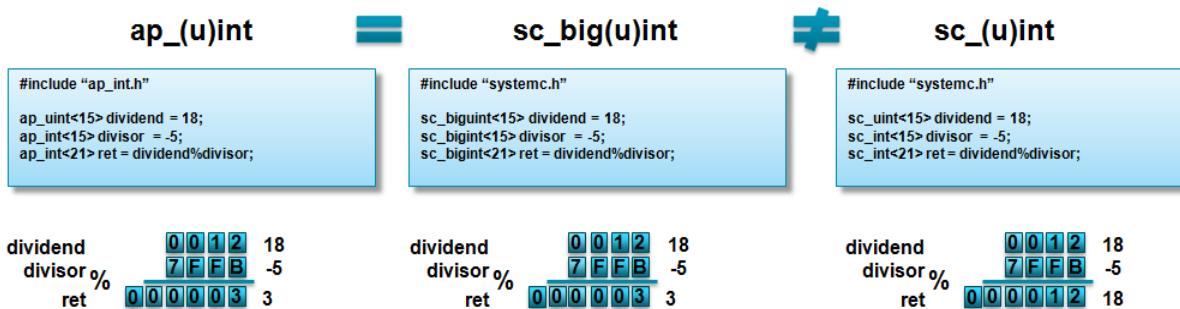


Slide 14-45:

Modulus Operator



- Vivado HLS tool is consistent with sc_biguint
 - sc_big(u)int can have an arbitrary bit length
- Both behave differently from sc_(u)int
 - sc_(u)int can only have a maximum of 64 bits



Slide 14-46:

Negative Shifts



- Vivado HLS tool ap_(u)int will shift in the opposite direction

- SystemC
 - sc_big(u)int types will not shift
 - sc_(u)int types will return a zero

ap_(u)int	sc_big(u)int	sc_(u)int																																																
<pre>#include "ap_int.h" ap_uint<15> op= 24; ap_int<15> shift= -2; ap_int<21> ret = op >> shift;</pre>	<pre>#include "systemc.h" sc_bignum<15> op= 24; sc_bignum<15> shift= -2; sc_bignum<21> ret = op >> shift;</pre>	<pre>#include "systemc.h" sc_uint<15> op= 24; sc_int<15> shift= -2; sc_int<21> ret = op >> shift;</pre>																																																
op shift >> ret <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>1</td><td>8</td></tr> <tr><td>7</td><td>F</td><td>F</td><td>E</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>6</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table> 24 -2 96	0	0	1	8	7	F	F	E	0	0	0	6	0	0	0	0	op shift >> ret <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>1</td><td>8</td></tr> <tr><td>7</td><td>F</td><td>F</td><td>E</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>8</td><td></td><td></td><td></td></tr> </table> 24 -2 24	0	0	1	8	7	F	F	E	0	0	0	1	8				op shift >> ret <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>1</td><td>8</td></tr> <tr><td>7</td><td>F</td><td>F</td><td>E</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table> 24 -2 0	0	0	1	8	7	F	F	E	0	0	0	0	0	0	0	0
0	0	1	8																																															
7	F	F	E																																															
0	0	0	6																																															
0	0	0	0																																															
0	0	1	8																																															
7	F	F	E																																															
0	0	0	1																																															
8																																																		
0	0	1	8																																															
7	F	F	E																																															
0	0	0	0																																															
0	0	0	0																																															

Slide 14-47:

Over-shift Left



- Vivado HLS tool ap_(u)int will shift then assign
 - Upper bits are lost
- SystemC both sc_big(u)int and sc_(u)int will assign then shift
 - Upper bits are saved

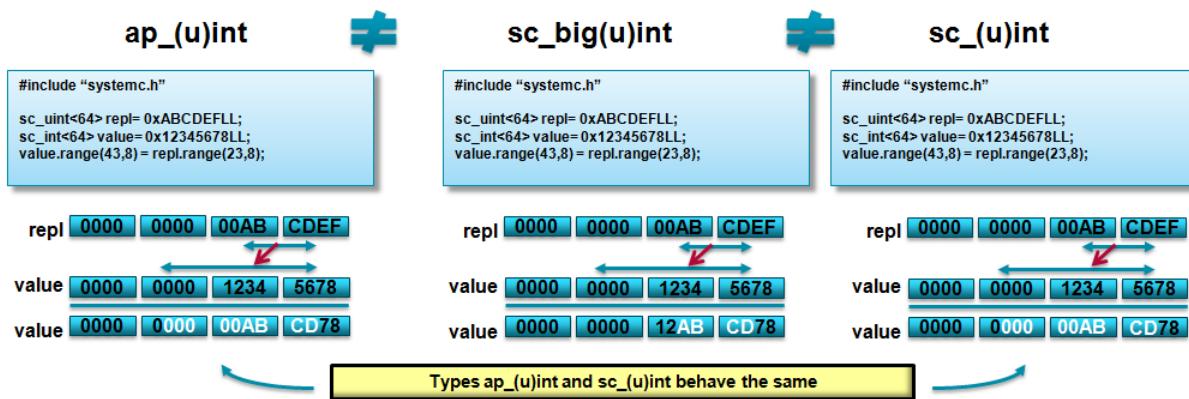
ap_(u)int	sc_big(u)int	sc_(u)int																																																
<pre>#include "ap_int.h" ap_uint<15> op= 0x7234; ap_int<15> shift= 4; ap_int<21> ret = op << shift;</pre>	<pre>#include "systemc.h" sc_bignum<15> op= 0x7234; sc_bignum<15> shift= 4; sc_bignum<21> ret = op << shift;</pre>	<pre>#include "systemc.h" sc_uint<15> op= 0x7234; sc_int<15> shift= 4; sc_int<21> ret = op << shift;</pre>																																																
op shift << ret <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>7</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>0</td><td>0</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>0</td><td>0</td><td>0</td></tr> </table> 29236 4 9024	7	2	3	4	0	0	0	4	0	0	2	3	4	0	0	0	op shift << ret <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>7</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>0</td><td>7</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>0</td><td>0</td><td>0</td></tr> </table> 29236 4 467776	7	2	3	4	0	0	0	4	0	7	2	3	4	0	0	0	op shift << ret <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>7</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>4</td></tr> <tr><td>0</td><td>7</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>0</td><td>0</td><td>0</td></tr> </table> 29236 4 467776	7	2	3	4	0	0	0	4	0	7	2	3	4	0	0	0
7	2	3	4																																															
0	0	0	4																																															
0	0	2	3																																															
4	0	0	0																																															
7	2	3	4																																															
0	0	0	4																																															
0	7	2	3																																															
4	0	0	0																																															
7	2	3	4																																															
0	0	0	4																																															
0	7	2	3																																															
4	0	0	0																																															

Slide 14-48:

Range Operation



- Vivado HLS tool ap_(u)int will replace and extend to fill the target range with zeros
- SystemC
 - sc_big(u)int will only update with the range of the source
 - sc_(u)int behaves the same as ap_int types

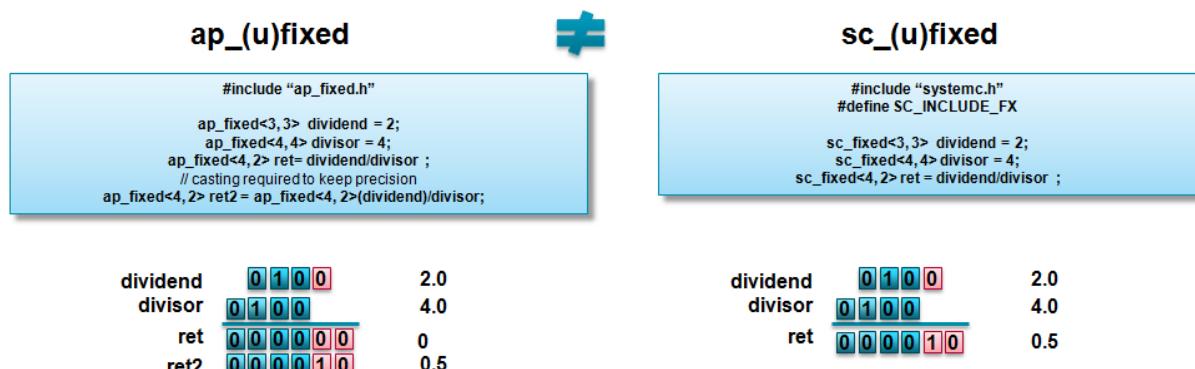


Slide 14-49:

Division and Fixed-Point Types



- ap_(u)fixed fraction is no greater than that of the dividend
 - Cast to explicitly extend the fraction part of the dividend
- SystemC sc_(u)fixed retains the precision on divide



Slide 14-50:

Right Shift and Fixed-Point Types



- Vivado HLS tool ap_(u)fixed shifts and then assigns
 - Quantization mode makes no difference to this behavior
- SystemC sc_(u)fixed will assign and then shift
 - Preserving redundant bits when greater quantization is used

ap_(u)fixed**sc_(u)fixed**

```
#include "ap_fixed.h"
ap_fixed<5,3,AP_RND,AP_SAT> val = 3.75
ap_fixed<5,3,AP_RND,AP_SAT> res = val >> 2;
ap_fixed<7,3,AP_RND,AP_SAT> res2 = val >> 2;
```

```
#include "systemc.h"
#define SC_INCLUDE_FX

sc_fixed<5,3,AP_RND,AP_SAT> val = 3.75
sc_fixed<5,3,AP_RND,AP_SAT> res = val >> 2;
sc_fixed<7,3,AP_RND,AP_SAT> res2 = val >> 2;
```

val		3.75
res		0.75
res2		0.75

val		3.75
res		0.75
res2		0.9375

Slide 14-51:

Left Shift and Fixed-Point Types



- Vivado HLS tool ap_(u)fixed sign extends the first operand then shifts and assigns
 - Quantization mode makes no difference to this behavior
- SystemC sc_(u)fixed will assign and then shift
 - No sign extension

ap_(u)fixed**sc_(u)fixed**

```
#include "ap_fixed.h"
ap_fixed<5,3,AP_RND,AP_SAT> val = 3.75
ap_fixed<5,3,AP_RND,AP_SAT> res = val << 2;
ap_fixed<7,5,AP_RND,AP_SAT> res2 = val << 2;
```

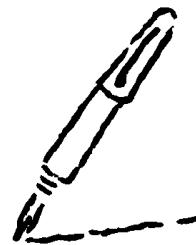
```
#include "systemc.h"
#define SC_INCLUDE_FX

ap_fixed<5,3,AP_RND,AP_SAT> val = 3.75
ap_fixed<5,3,AP_RND,AP_SAT> res = val << 2;
ap_fixed<7,5,AP_RND,AP_SAT> res2 = val << 2;
```

val		3.75
res		-1
res2		-1

val		3.75
res		3.75
res2		15

Capture Your Notes Here



Appendix

Additional Xilinx Courses

For detailed information on any of the following courses, go to www.xilinx.com/training.

Embedded Design

Hardware:

- Zynq UltraScale+ MPSoC for the Hardware Designer
- Introduction to the Zynq All Programmable SoC Architecture
- Embedded Systems Design
- Advanced Features and Techniques of Embedded Systems Design

Software:

- Embedded C/C++ SDSoc Development Environment and Methodology
- Zynq UltraScale+ MPSoC for the Software Developer
- Embedded Systems Software Design
- Advanced Features and Techniques of Embedded Systems Software Design
- Embedded Design with PetaLinux Tools

System Architect:

- Zynq All Programmable SoC System Architecture
- Zynq UltraScale+ MPSoC for the System Architect

HLx and SDx Design

HLx:

- C-based Design: High-Level Synthesis with Vivado HLS
- C-based HLS Coding for Hardware Designers
- C-based HLS Coding for Software Designers

SDx:

- Embedded C/C++ SDSoc Development Environment and Methodology

Architecture

FPGAs:

- Designing with the UltraScale Architecture (two day)
- UltraScale Architecture Workshop (one day)
- Designing with the 7 Series Families
- Designing with the Spartan-6 and Virtex-6 Families
- Designing with the Virtex-5 Family

SoCs and MPSoCs:

- Zynq All Programmable SoC System Architecture
- Zynq UltraScale+ MPSoC for the System Architect

FPGA Design

Vivado Design Suite:

- Designing FPGAs Using the Vivado Design Suite 1
- Designing FPGAs Using the Vivado Design Suite 2
- Designing FPGAs Using the Vivado Design Suite 3
- Designing FPGAs Using the Vivado Design Suite 4
- Xilinx Partial Reconfiguration Tools and Techniques

ISE Design Suite:

- ISE Design Tool Flow
- Essentials of FPGA Design
- Designing for Performance
- Advanced FPGA Implementation

Vivado Design Suite Adopter:

- UltraFast Design Methodology
- Vivado Design Suite for ISE Software Project Navigator Users
- Vivado Design Suite Advanced XDC and Static Timing Analysis for ISE Software Users

Connectivity Design

PCI Express Protocol:

- PCIe Protocol Overview
- Designing an Integrated PCI Express System

Transceivers:

- Designing with Multi-Gigabit Serial I/O
- Designing with UltraScale FPGA Transceivers

Memory Interface:

- How to Design a High-Speed Memory Interface

Ethernet MAC:

- Designing with Ethernet MAC Controllers

Signal Integrity and Board Design:

- Signal Integrity and Board Design for Xilinx FPGAs

DSP Design

- Essential DSP Implementation Techniques for Xilinx FPGAs
- DSP Design Using System Generator

Languages

SystemVerilog:

- Designing with SystemVerilog
- Verification with SystemVerilog

Verilog:

- Designing with Verilog

VHDL:

- Designing with VHDL
- Advanced VHDL