



# Vivado HLx Design Entry

June 2016

# Agenda

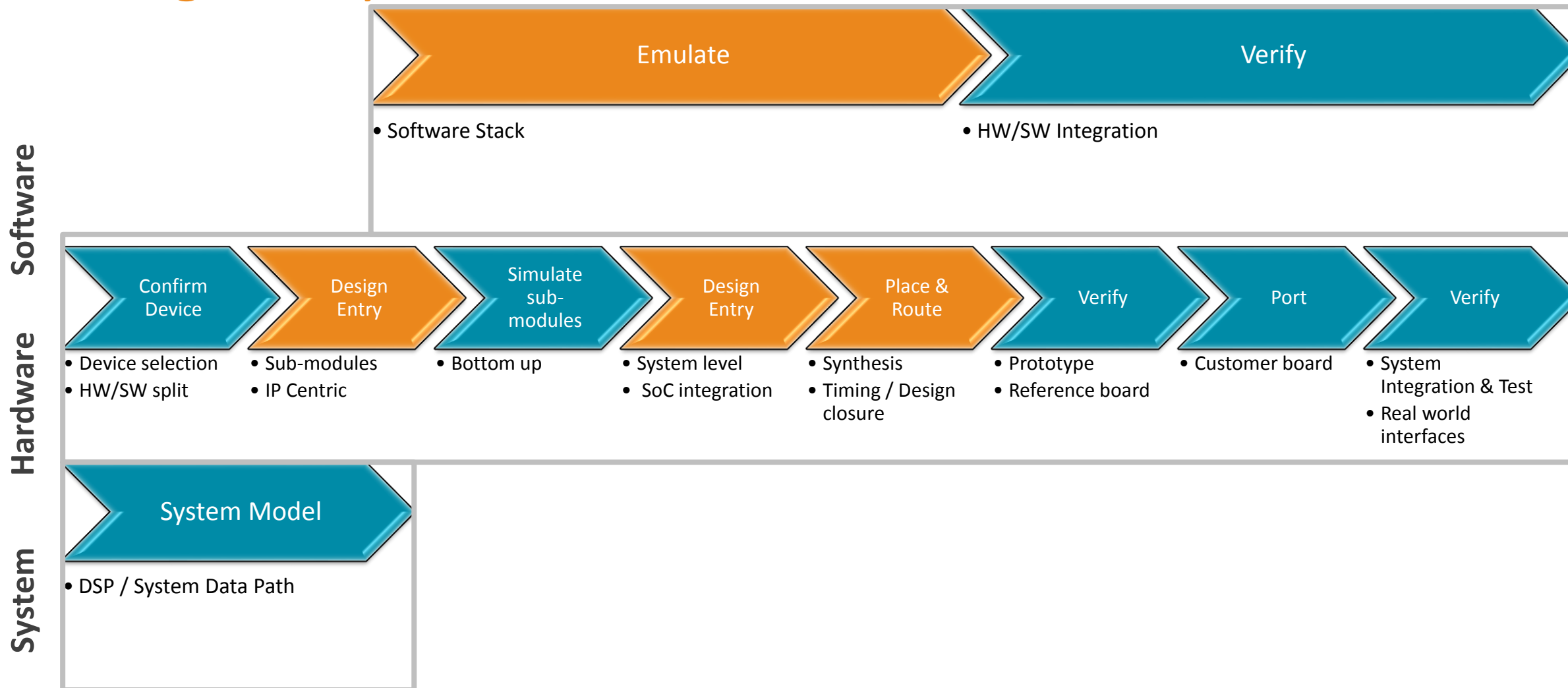
- What is the HLx Design Methodology?
- New & Early Access features for Connectivity Platforms
- Creating Differentiated Logic

# What is the HLx Design Methodology?



# Bottom Up – SoC Design (not HLx Flow)

## Design Entry – Verification



# Why We Need HLx System Design Methodology

- **NRE (design cost) is considered an issue for All Programmable SoCs**

- Bottleneck for Hardware and Software integration

- **Physical Connectivity & IP use can be time consuming**

- Many customers estimate at 40%+ total effort

- **Fewer FPGA capable engineers vs. software team**

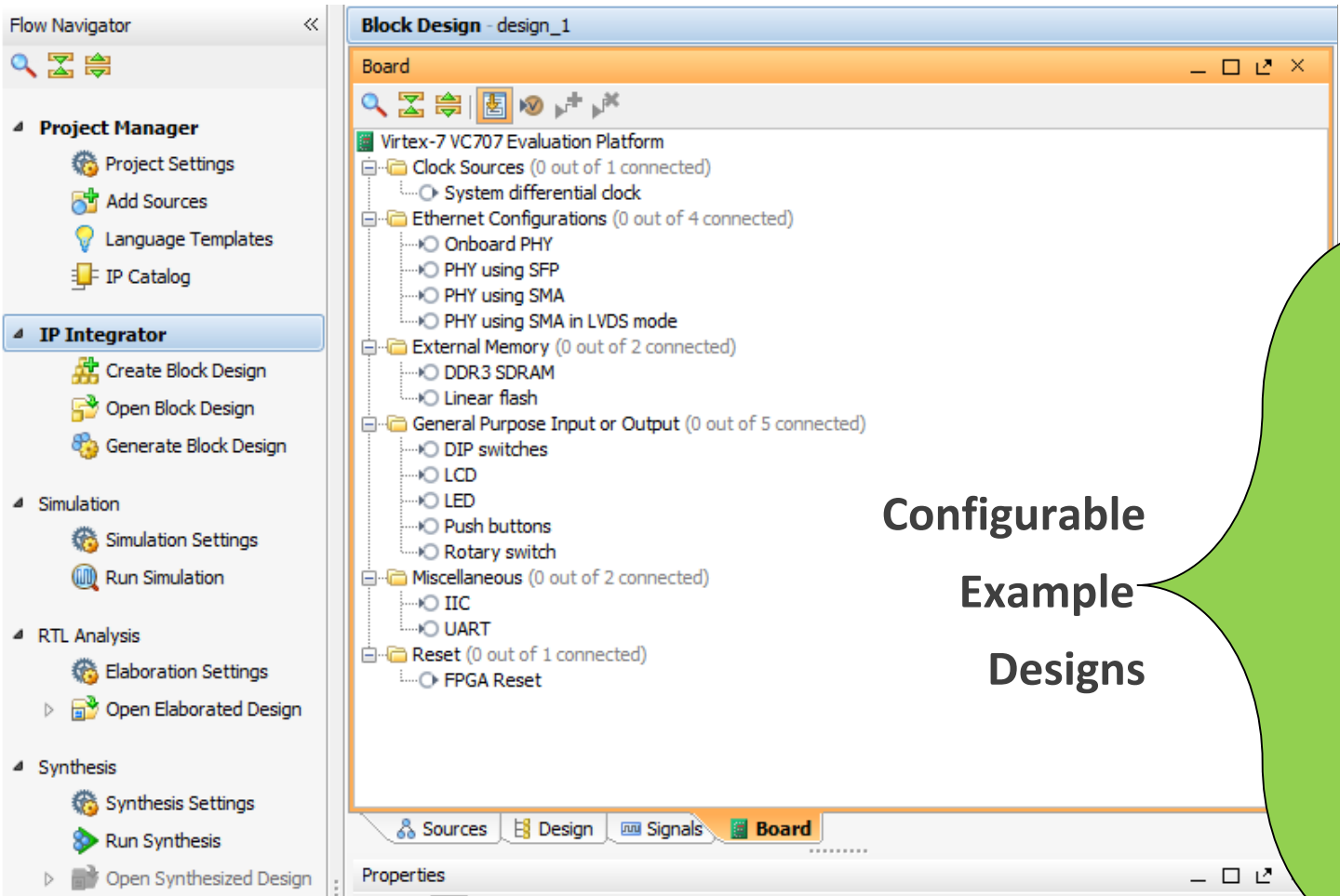


# What Makes HLx a Higher Level Methodology?

## 4 Steps to Solving SoC Design for the FPGA

1. Separate the **connectivity platform** and the **differentiated logic**
2. Utilize Vivado High-Level Synthesis and C/C++ libraries to design the differentiated logic
3. Exhaustively test the differentiated logic algorithm through C-based simulation
4. Automate rapid configuration, generation and closure of the connectivity platform, then incorporate the differentiated logic in IP Integrator

# Designer Assistance for the Connectivity Platform



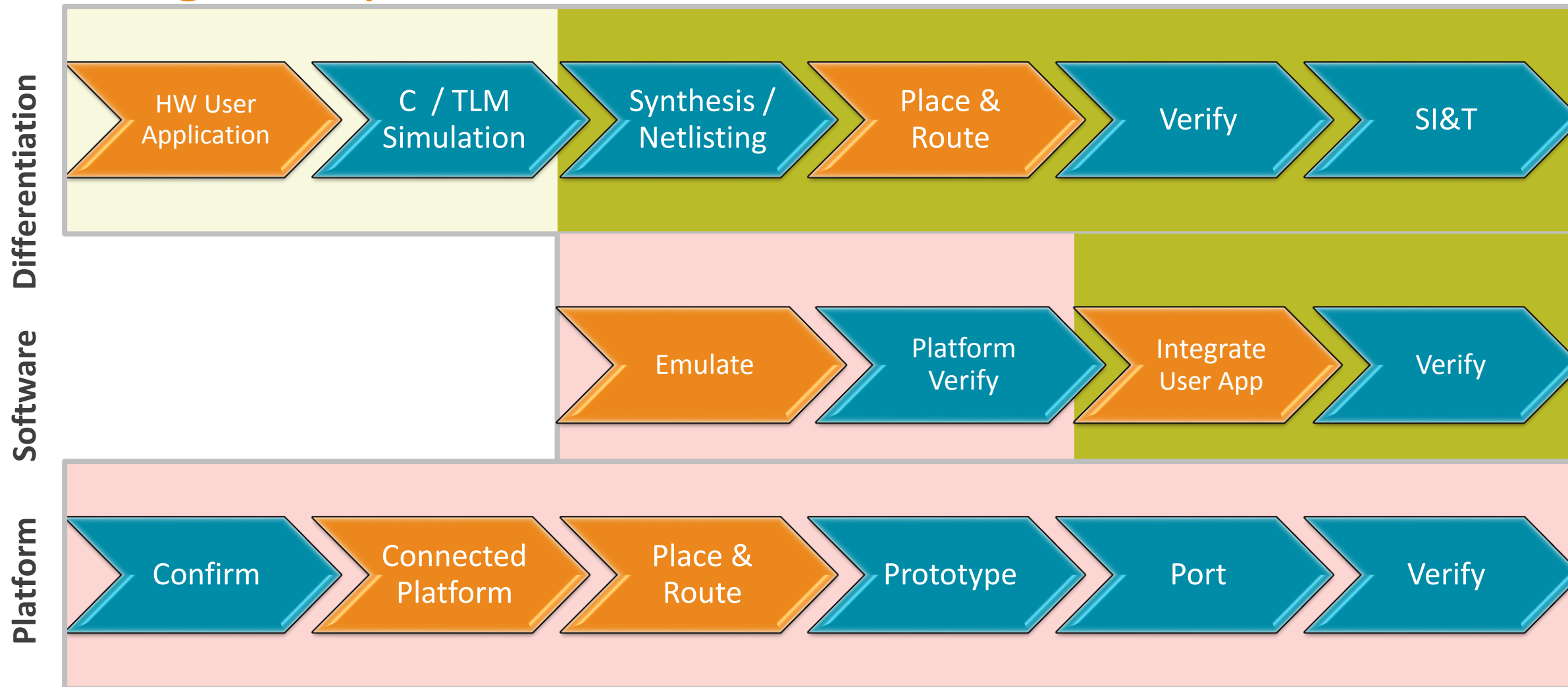
Configurable  
Example  
Designs

Layer	Example	Automation
Application	User code	Examples / Linux
Presentation	Drivers	HW Handoff
Session	IP Addresses	Connection
Transport	AXI MM	Connection
Network	AXI Interconnect	Connection
MAC	Memory controller	Block
PHY	DDR I/F	Board



# Bottom Up – HLx Based Methodology

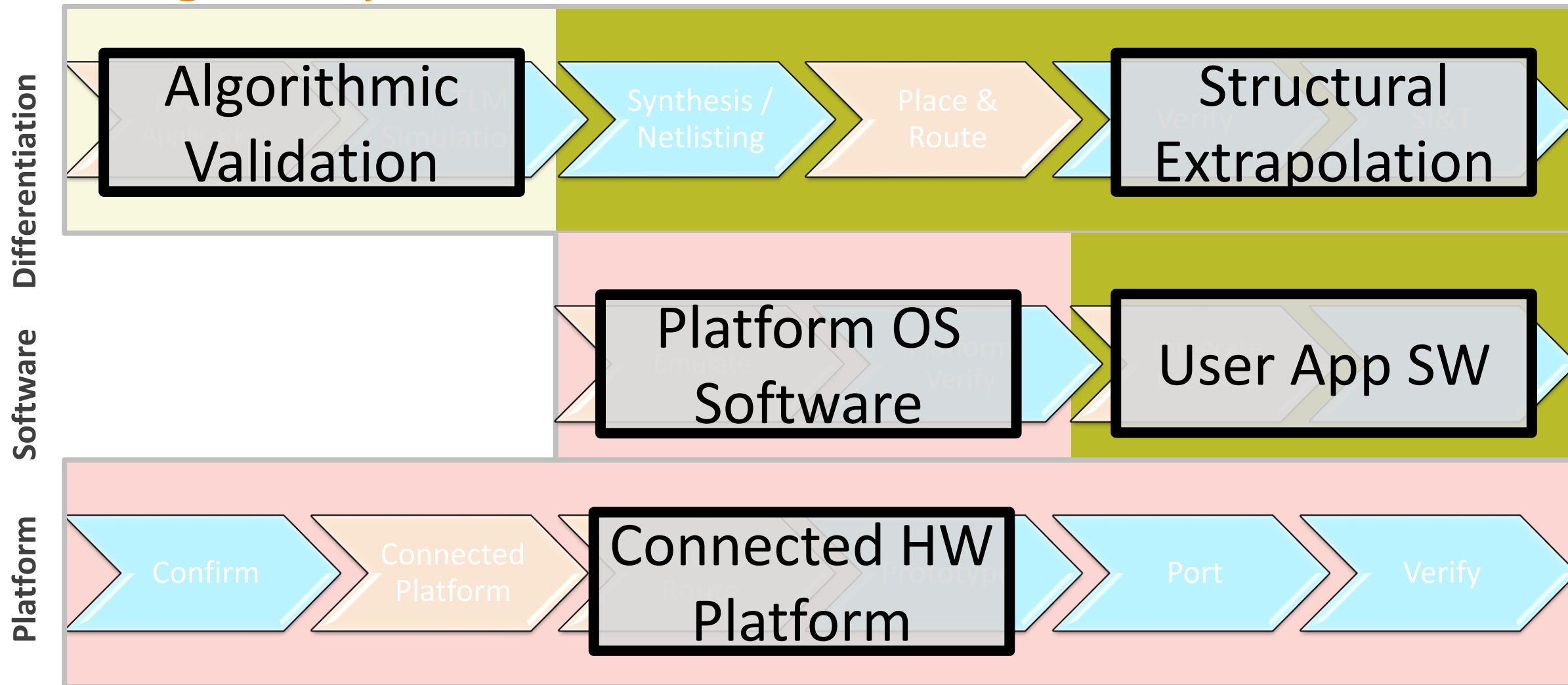
## Design Entry – Verification





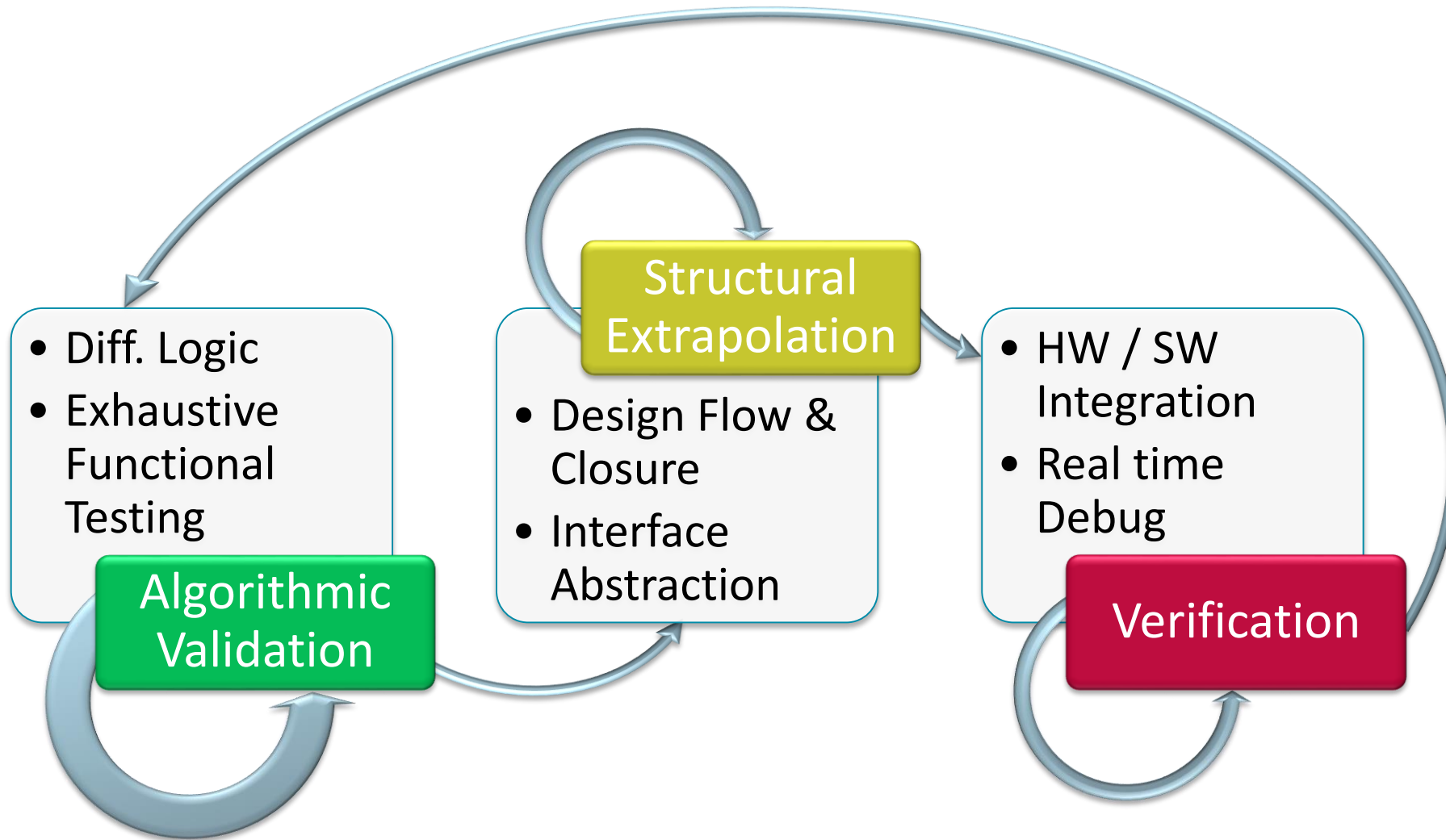
# Bottom Up – HLx Based Methodology

## Design Entry – Verification



# The HLx Differentiated Logic Design Flow

## What makes it faster?



Actual example:

### Traditional Flow

- **240 people\*mo**
  - 10 people
  - 2 years

**15x faster**

### HLS Based Flow

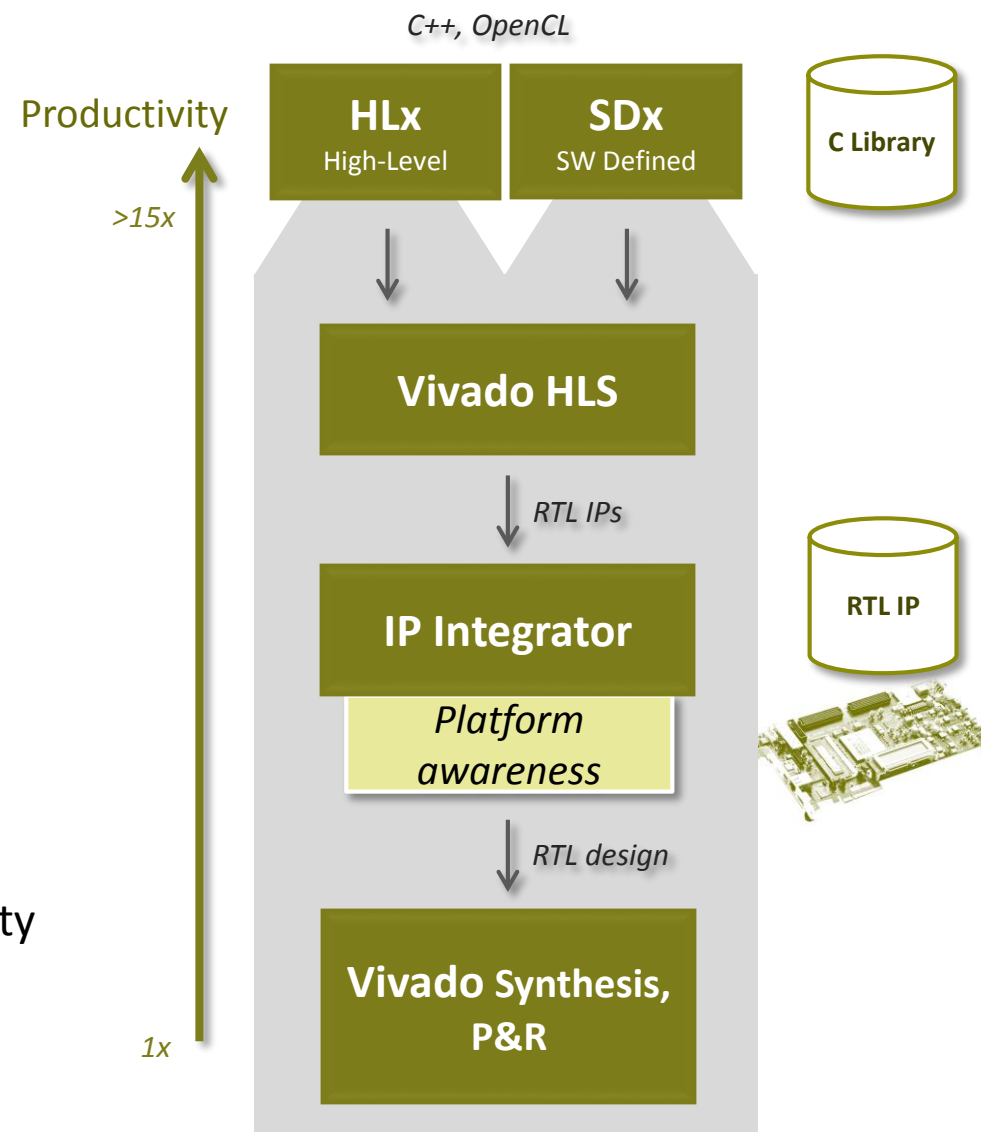
- **16 people\*mo**
  - 2 people
  - 8 month

**Faster for derivative designs**

- C++ reuse
- Scales with parameters
- Device independent

# HLx Summary – Accelerating Design Productivity

- **Separate platform design from differentiated logic**
  - Let application designers focus on the differentiated logic
- **Spend less time on the standard connectivity**
  - **IPI**: configure & generate a platform on a custom board
  - Use of Partial Reconfiguration to guarantee performance
- **Spend more time on the differentiated logic**
  - **HLS**: enabling core technology: C/C++/OpenCL synthesis
    - Exhaustive simulation, architecture exploration, code portability
  - HLx: Accelerates HW design: IP design (HLS/SysGen) + connectivity platform integration (IP Integrator)
  - SDx: Brings SW programmability to FPGA based platform



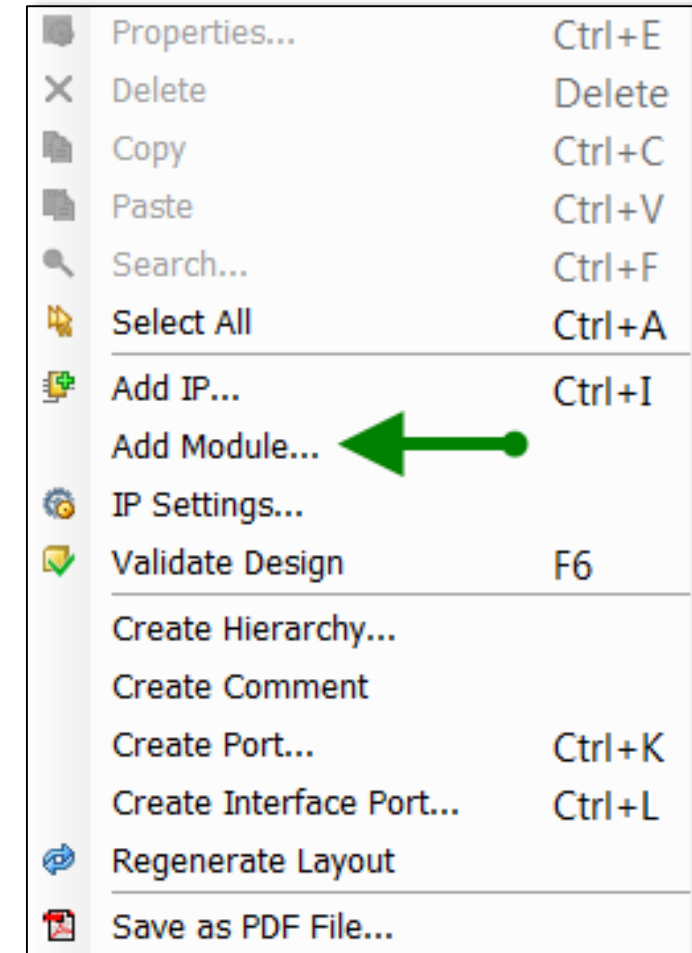
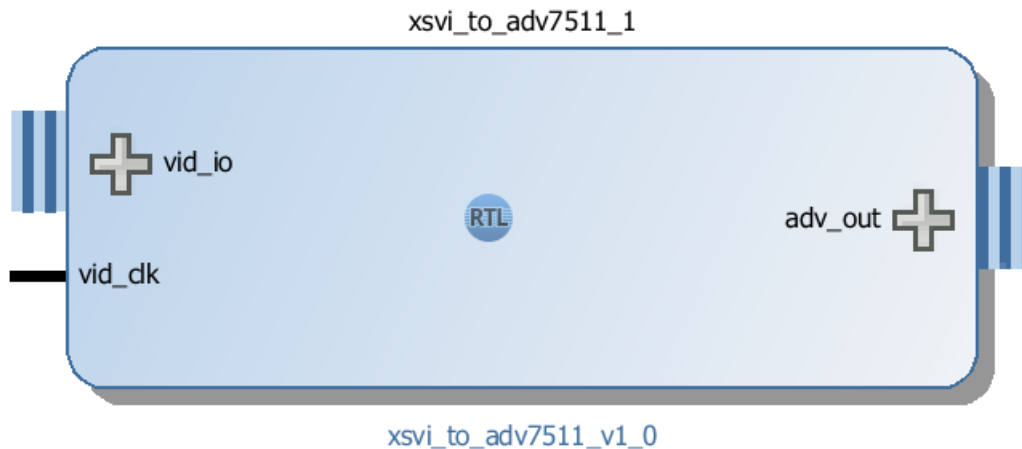
# New & Early Access features for Connectivity Platforms



# Add Module (New)

## ➤ New “HDL Module Reference” flow

- Easily add HDL to a BD without using the IP Packager
- AXI Interfaces are automatically inferred
- Source files can live outside of the project
- Access via right-click in sources tree or on the canvas



# Add Module Does / Does Not

## ➤ Does

- Allow you to add hierarchical modules
- Auto infer standard AXI Interfaces
- Allow custom interfaces to be used
- Allow busif\_association to be specified
- Allow reset polarity to be specified
- Pass generics / parameters through XGUI
- Work on the original source files

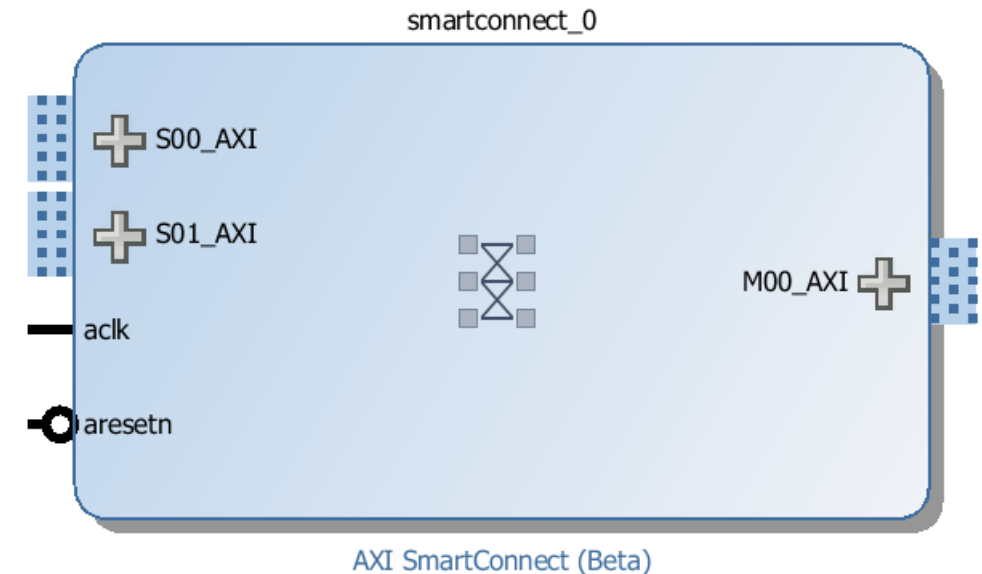
## ➤ Does Not

- Allow you to add hierarchies with IP cores
- Allow you to add non-RTL hierarchies
- Give you a packaged IP for reuse
- Make it easy to modify lower hierarchical RTL

# AXI SmartConnect (Early Access)

## ➤ Early Access release of AXI SmartConnect IP

- Optimized for high-performance systems
- Will replace the AXI Interconnect in the future
- Simplified clocks and reset
- No cost early access license required for use in 2016.1



## ➤ Disk size / Synthesis runtime issues in 2016.1

- To be fixed in Vivado 2016.3

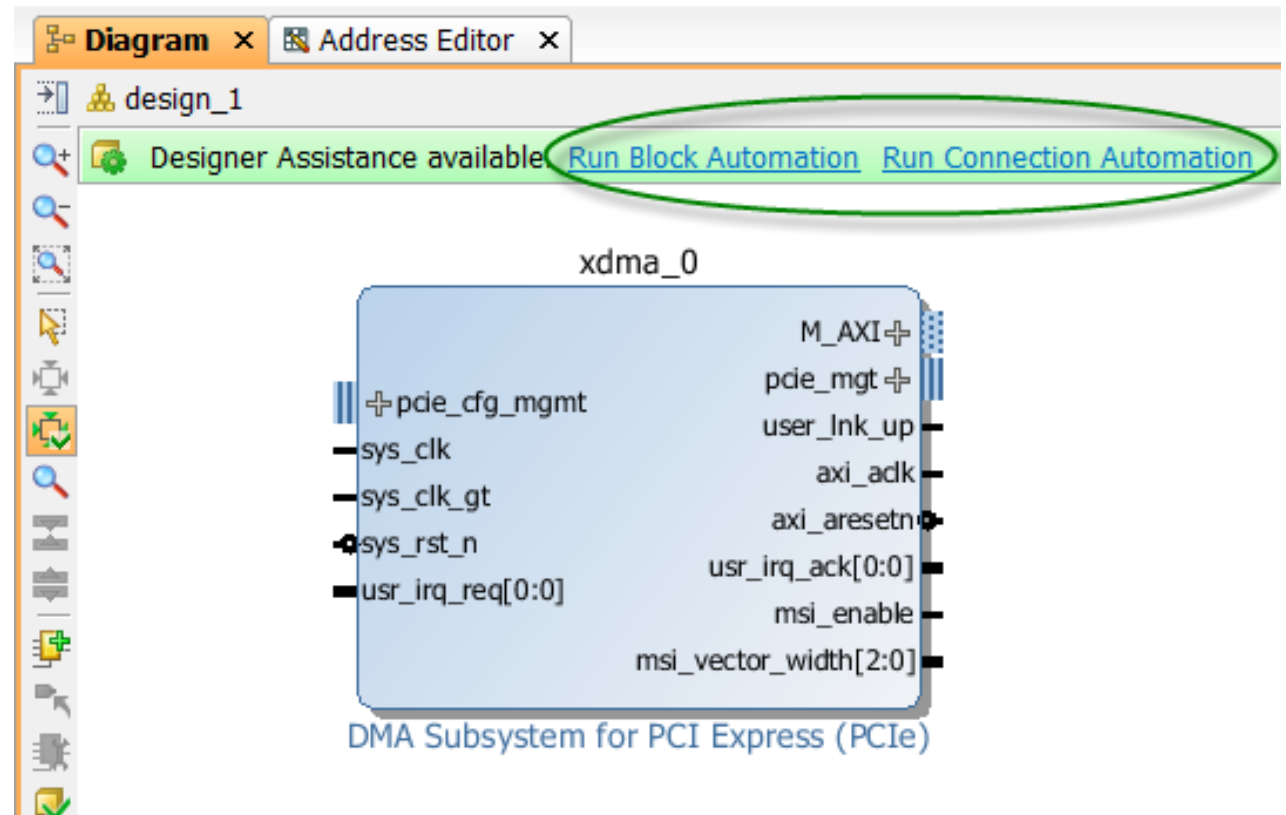
➤ Request licenses – email your Xilinx licensing ID to [smartconnect@Xilinx.com](mailto:smartconnect@Xilinx.com)



# PCIe Designer Assistance (New)

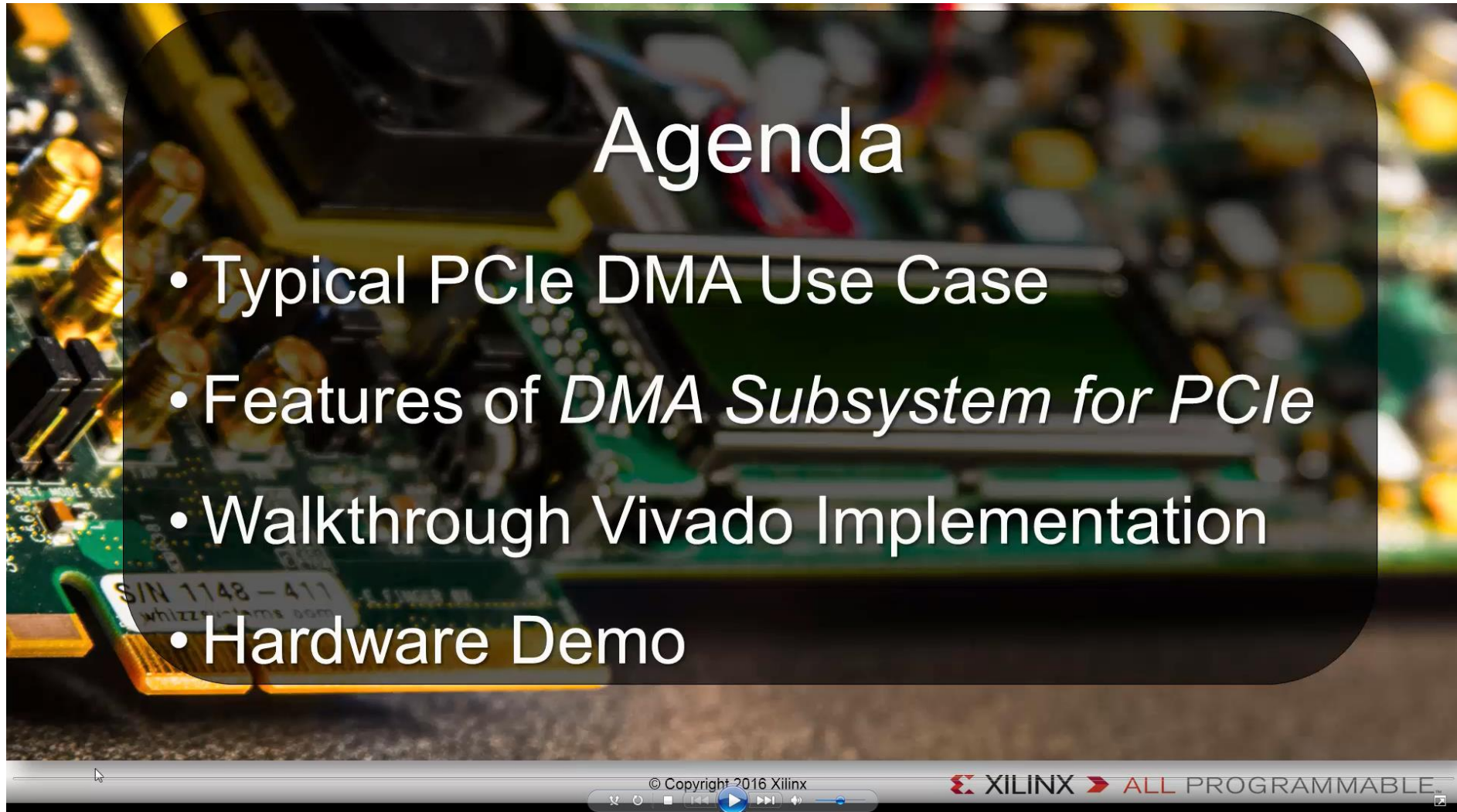
## ➤ Enhanced PCIe Designer Assistance for these boards:

- Alpha-Data ADM-PCIE-7V3
- Kintex UltraScale Alpha-Data board
- Kintex UltraScale KCU105 Evaluation Platform
- Virtex-7 VC709 Evaluation Platform
- Virtex UltraScale VCU108 Evaluation Platform



# PCIe Designer Automation, how can I really use this?

## Check out the Quick Take Video!



**Agenda**

- Typical PCIe DMA Use Case
- Features of *DMA Subsystem for PCIe*
- Walkthrough Vivado Implementation
- Hardware Demo

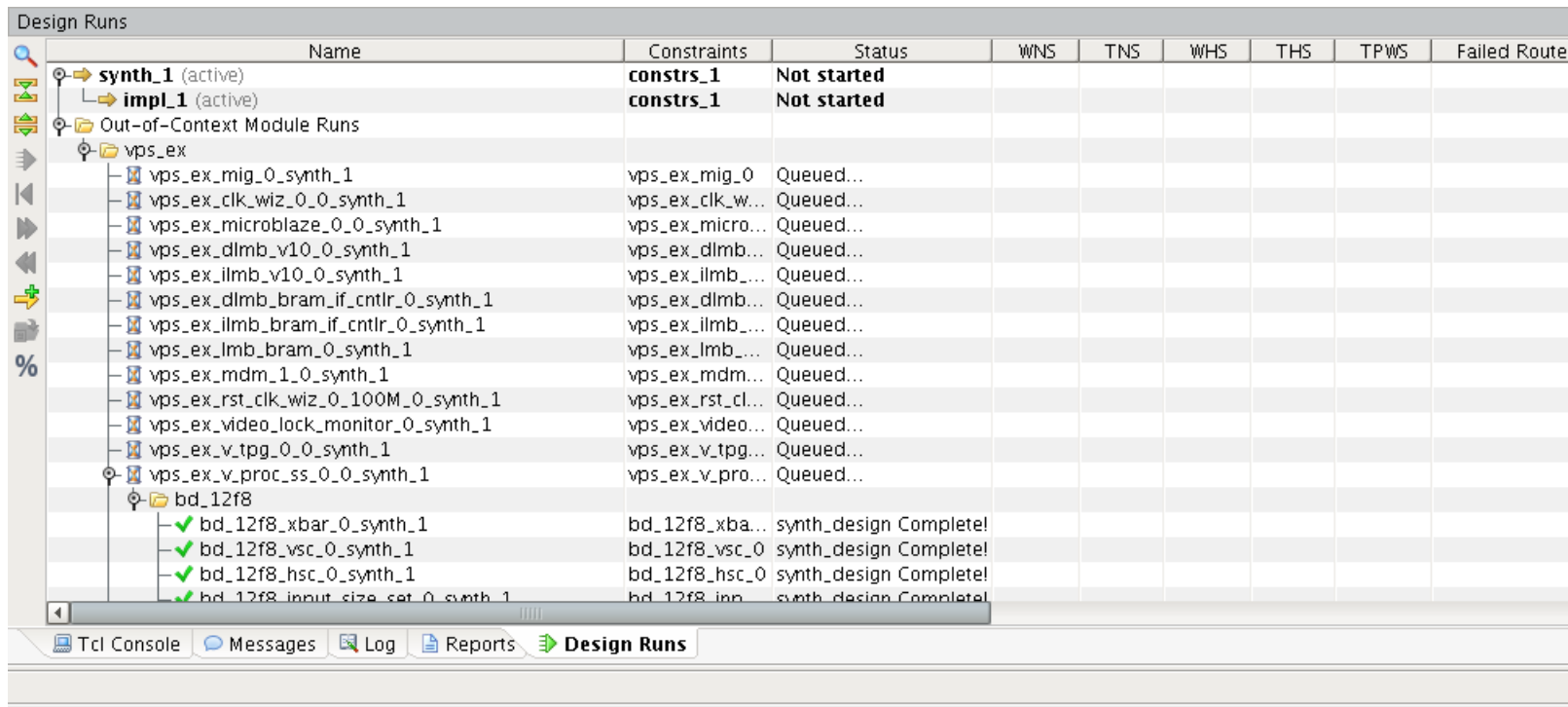
© Copyright 2016 Xilinx

XILINX ALL PROGRAMMABLE

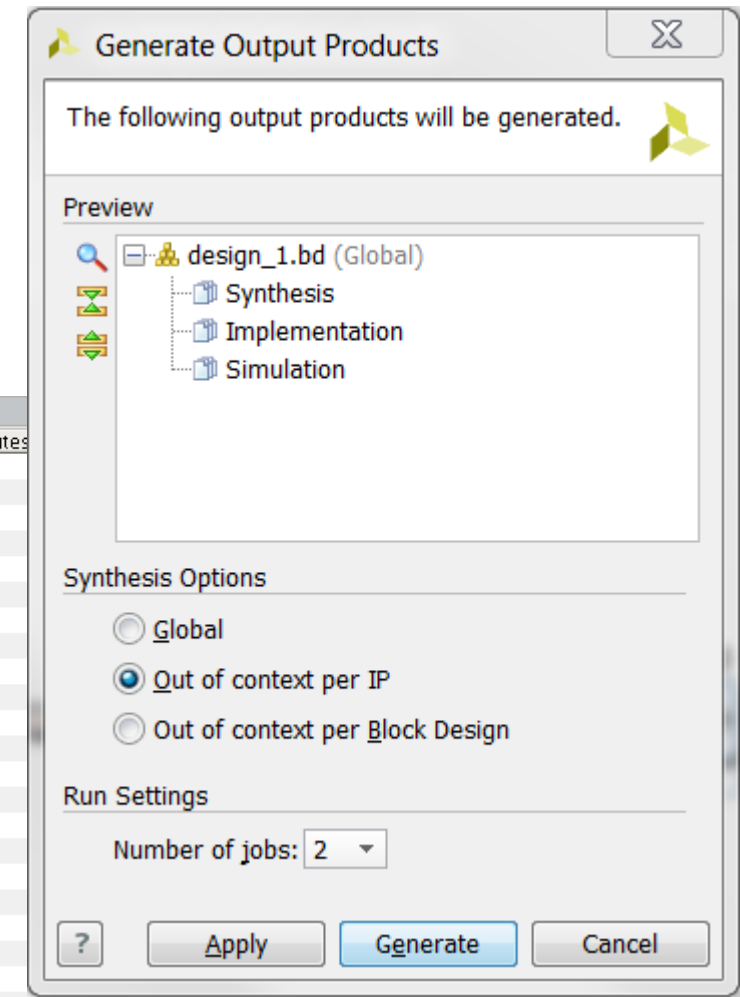
# Core Caching & OOC per IP

## ► Improved caching of IP during BD synthesis

- Reduced memory usage to determine a hit
- Hit times reduced up to 100x [0.5-4.0 sec in 2016.1 vs 45-60 seconds in 2015.3]



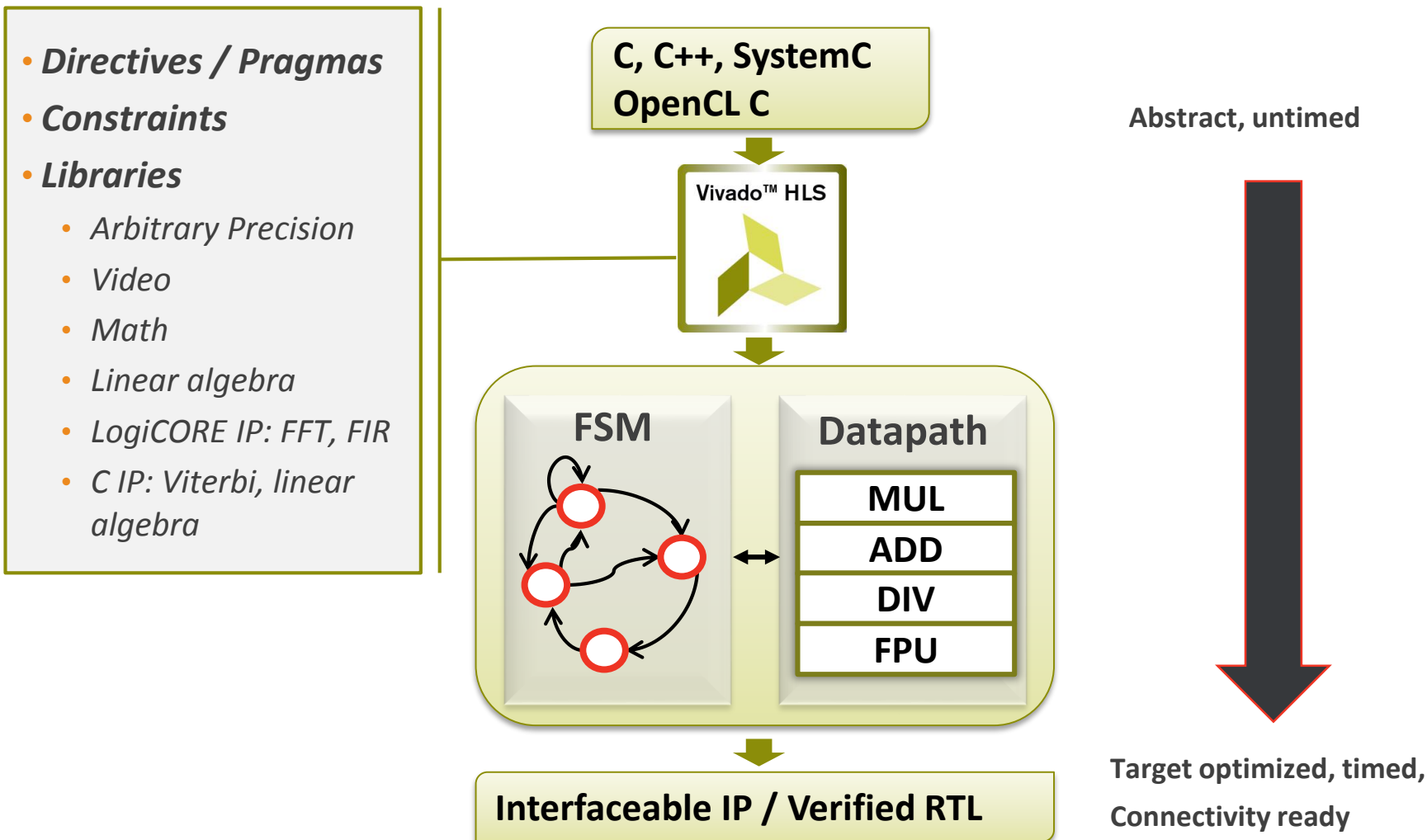
Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Failed Routes
synth_1 (active)	constrs_1	Not started						
impl_1 (active)	constrs_1	Not started						
Out-of-Context Module Runs								
vps_ex								
vps_ex_mig_0_synth_1	vps_ex_mig_0	Queued...						
vps_ex_clk_wiz_0_0_synth_1	vps_ex_clk_wiz_0_0	Queued...						
vps_ex_microblaze_0_0_synth_1	vps_ex_microblaze_0_0	Queued...						
vps_ex_dlm_b_v10_0_synth_1	vps_ex_dlm_b_v10_0	Queued...						
vps_ex_ilmb_v10_0_synth_1	vps_ex_ilmb_v10_0	Queued...						
vps_ex_dlm_b_bram_if_cntlr_0_synth_1	vps_ex_dlm_b_bram_if_cntlr_0	Queued...						
vps_ex_ilmb_bram_if_cntlr_0_synth_1	vps_ex_ilmb_bram_if_cntlr_0	Queued...						
vps_ex_lmb_bram_0_synth_1	vps_ex_lmb_bram_0	Queued...						
vps_ex_mdm_1_0_synth_1	vps_ex_mdm_1_0	Queued...						
vps_ex_rst_clk_wiz_0_100M_0_synth_1	vps_ex_rst_clk_wiz_0_100M_0	Queued...						
vps_ex_video_lock_monitor_0_synth_1	vps_ex_video_lock_monitor_0	Queued...						
vps_ex_v_tpg_0_0_synth_1	vps_ex_v_tpg_0_0	Queued...						
vps_ex_v_proc_ss_0_0_synth_1	vps_ex_v_proc_ss_0_0	Queued...						
bd_12f8								
bd_12f8_xbar_0_synth_1	bd_12f8_xbar_0	synth_design Complete!						
bd_12f8_vsc_0_synth_1	bd_12f8_vsc_0	synth_design Complete!						
bd_12f8_hsc_0_synth_1	bd_12f8_hsc_0	synth_design Complete!						
bd_12f8_input_size_set_0_synth_1	bd_12f8_input_size_set_0	synth_design Complete!						



# Creating Differentiated logic

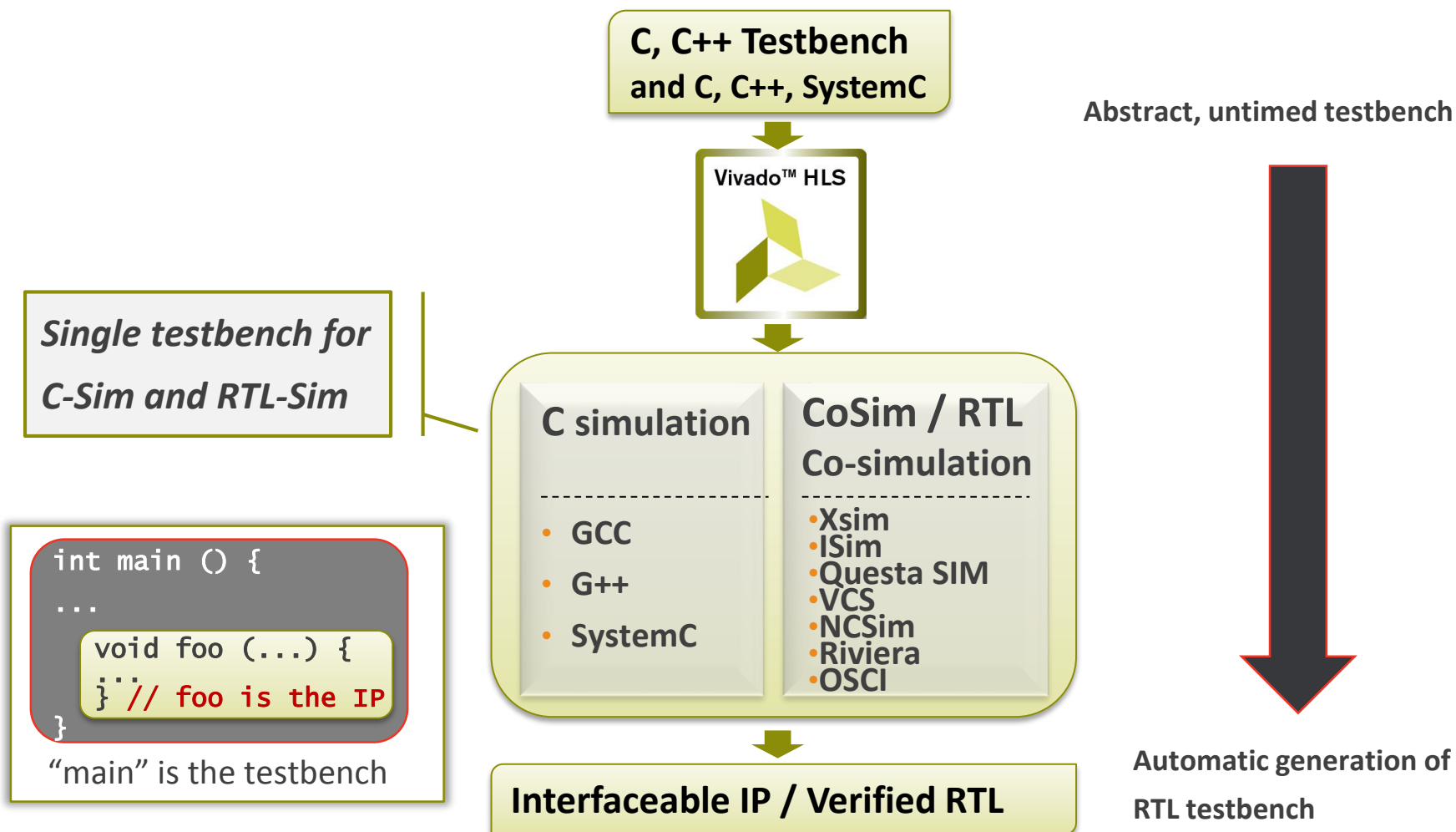


# Vivado HLS – Synthesis



Accelerates C to RTL Creation

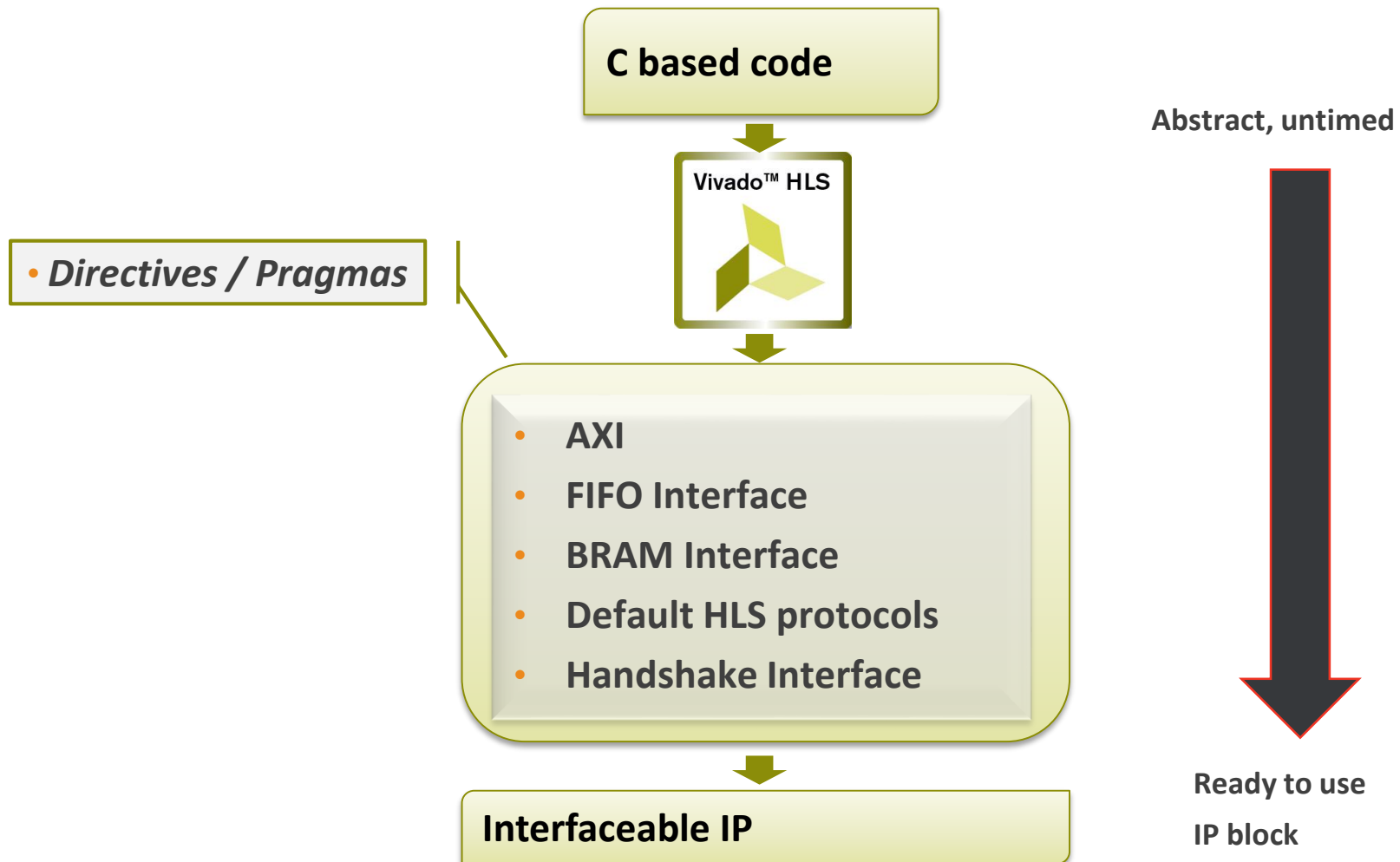
# Vivado HLS - Verification



Accelerates C to RTL Creation



# Vivado HLS - Interfaces

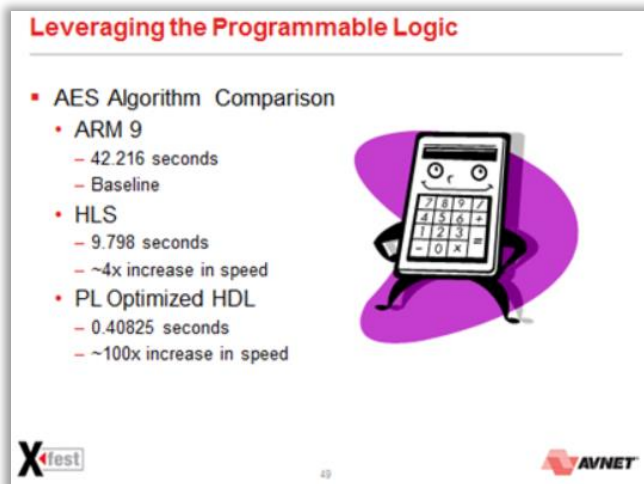


Accelerates Algorithmic C to RTL Creation



# UltraFast For Maximum Throughput

- By default, the compiled C code will typically have a large latency
  - Loops can incur a very substantial amount of latency (can be changed using pragmas)
  - Pipeline and Dataflow are commonly used
- C code is likely to require some re-write
  - To allow for maximum throughput
- Un-optimized settings and C code won't show as good as RTL



## Benchmark Example (a little misleading):

AES algorithm show HLS default performance 4X faster than ARM but 20X slower than RTL.

This can be expected since by default the latency of HLS will be high. HLS can only compete with RTL if the UltraFast methodology steps are followed.

# Expression Balance Impact on QoR – FIR

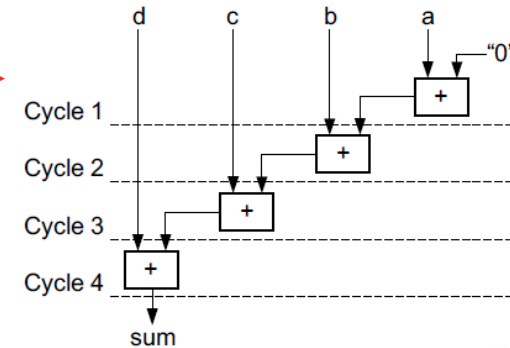
## ➤ DSP blocks can be summed as adder tree or adder cascade

– FIR calculation (*result += x[i]\*coef[i]*) gets re-arranged as a balanced tree...

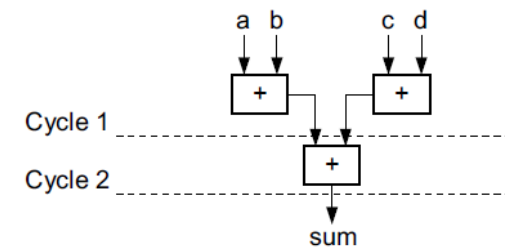
```
void fir (data_y *y, coef_t c[N], data_t x) {  
    static data_t shift_reg[N];  
    data_y acc;  
    acc=0;  
  
    for (int i=N-1;i>=0;i--) {  
        if (i==0) {  
            acc+=x*c[0];  
            shift_reg[0]=x;  
        } else {  
            shift_reg[i] = shift_reg[i-1];  
            acc+=shift_reg[i]*c[i];  
        }  
    }  
    *y=acc;  
}
```

*FIR Description (a cascade chain of multiply / add)*

*Compiler interpretation*



*Balanced optimization*



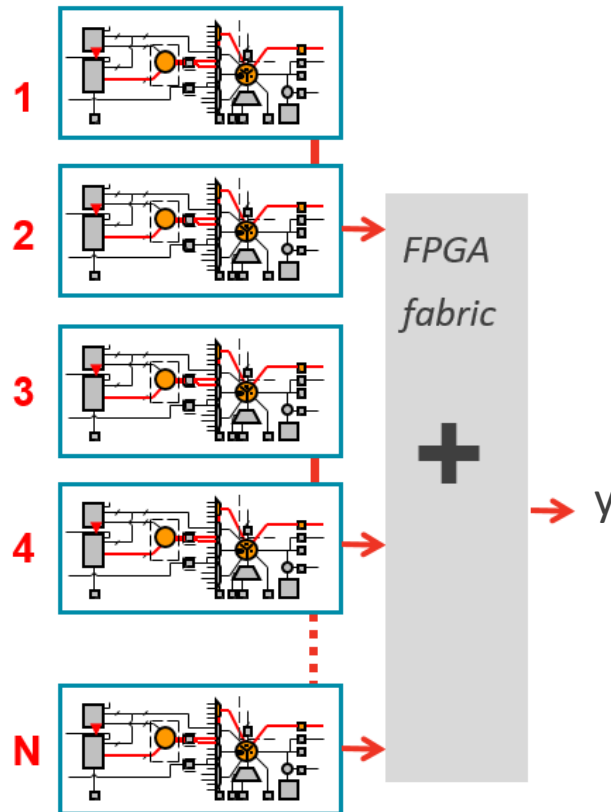
# Expression Balance Impact on QoR – FIR (Example)

## ➤ Expression balance off saves area (adder cascade)

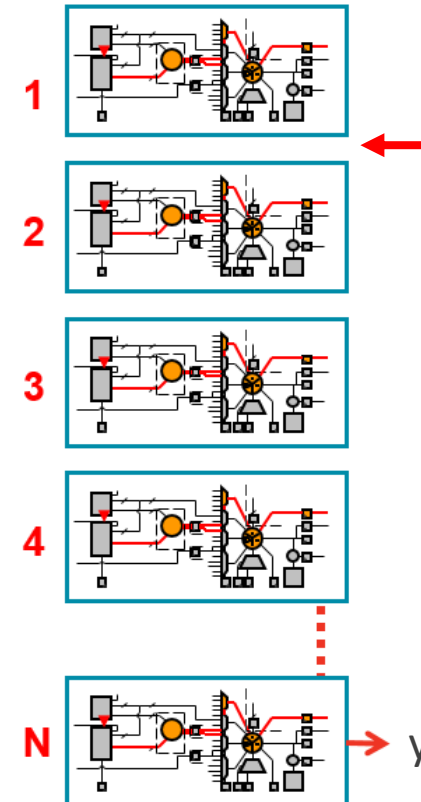
– But increases latency

*Default balancing of adders*

250 LUTs  
607 MHz



*With expression balance pragma OFF*



DSP blocks implement an  
adder cascade  
No fabric logic necessary.

37 LUTs  
666 MHz

# UltraFast Methodology



➤ See details in UG1197 (Chapter 4)

– The UltraFast High-Level Productivity Design Methodology Guide (accessible from the Design Hub)

## 1: Add Initial Directives

- Define interfaces (and data packing)
- Define loop trip counts

## 2: Pipeline

- Pipeline and Dataflow

## 3: Improve Pipelining

- Partition memories and ports
- Remove false dependencies

## 4: Improve Area

- Optionally recover resources through sharing

## 5: Reduce Latency

- Tweak operator sharing and constraints

# Step 1: Add Initial Optimization Directives

- Define interfaces right at the beginning
- Size loop trip count to work with realistic numbers in the reports
- The “config\_\*” settings provide global directives

<i>Directives and Configurations</i>	<i>Description</i>
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 which have variables bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.
config_interface	Controls IO ports not associated with the top-level function arguments and allows unused ports to be eliminated from the final RTL.

# Step 2: Directives for Pipelined Designs

## Pipeline functions and loops

<i>Directives and Configurations</i>	<i>Description</i>
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.

# Step 3: Improve Pipelining

Reaching maximum throughput (II = 1)

➤ Typically bandwidth and dependencies

➤ Sometimes the C code itself prevents dataflow

<i>Directives and Configurations</i>	<i>Description</i>
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 for-loops to create multiple independent operations rather than a single collection of operations.
config_array_partition	This configuration 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



# Step 4: Improving Area

Typically performance is #1 but these can help reduce area

➤ **Use the Resource tab in the Analysis Perspective to see sharing**

<i>Directives and Configurations</i>	<i>Description</i>
ALLOCATION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks.
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.
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.

# Step 5: Reducing the latency

## ➤ Short latency could be a design requirement

–These directives help control latency...

<i>Directives and Configurations</i>	<i>Description</i>
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.

# Learn More about Vivado HLS



- Vivado HLS is included in all Vivado HLx Editions
- Videos on xilinx.com and YouTube
- Vivado HLS “**Design Hub**” in DocNav: Tutorials, UG, app notes, videos, etc...
- Code examples within the tool
- App notes on Xilinx.com (or linked from design hub)
- Instructor lead training

# Summary

1. Separate the **connectivity platform** and the **differentiated logic**
2. Utilize Vivado High-Level Synthesis and C/C++ libraries to design the differentiated logic
3. Exhaustively test the differentiated logic algorithm through C-based simulation
4. Automate rapid configuration, generation and closure of the connectivity platform, then incorporate the differentiated logic in IP Integrator