UltraFast High-Level Productivity Design Methodology Guide

UG1197 (v2018.3) December 5, 2018

Revision History

12/05/2018: Released with Vivado® Design Suite 2018.3 without changes from 2018.2.

Section	Revision Summary
06/06/2018 Version 2018.2	
Vivado HLS Design Flow	Updated the Vivado HLS Design Flow.

Table of Contents

Appendix A: Additional Resources and Legal Notices

Chapter 1

High-Level Productivity Design Methodology

About This Guide

Xilinx® programmable devices have capacities of multi-million Logic Cells (LC), and integrate an ever-increasing share of today's complex electronic systems. This High-Level Productivity Design Methodology provides a set of best practices to create such complex systems within short design cycles.

The methodology focuses on the following concepts:

- Using parallel development flows for the valuable differentiated logic which differentiates your products in the marketplace and the shell used to integrate the differentiated logic with the rest of the ecosystem.
- Extensive use of a C-based IP development flow for the differentiated logic to provide simulations that are orders of magnitude faster than RTL simulations, as well as accurately timed and optimized RTL.
- Use of existing pre-verified, block, and component-level IP to quickly build the shell which encapsulates your differentiated logic in the system.
- Use of scripts to highly automate the flow from accurate design validation through to programmed FPGA.

The recommendations in this guide have been gathered from a large pool of expert users over the past few years. They have consistently delivered the following improvements over traditional RTL design methodologies:

- 4X speed up in development time for designs.
- 10X speed up in the development time for derivative designs.
- 0.7X to 1.2X the Quality of Results (QoR).

Although this guide focuses on large complex designs, the practices discussed are suitable for, and have successfully been applied in, all types of design including:

- Digital Signal Processing:
  - Image processing
  - Video
  - Radar
- Automotive
- Processor acceleration
- Wireless
- Storage
- Control systems

Need for a New Design Methodology

The advanced designs used in today's increasingly complex electronic products are stretching the boundaries of density, performance, and power. They create a challenge for design teams to hit a target release window within their allocated budget.

A productive methodology for addressing these design challenges is one where more time is spent at higher levels of abstraction, where verification times are the fastest and productivity gains are the greatest.

The need for a new design methodology is highlighted in the following figure, where the area of each region represents the percentage of development effort at each stage of the design flow.

- With a traditional RTL methodology most of the effort is spent on the implementation details.
- In a high-level productivity design methodology, most of the effort is spent designing and verifying you are building the right system.

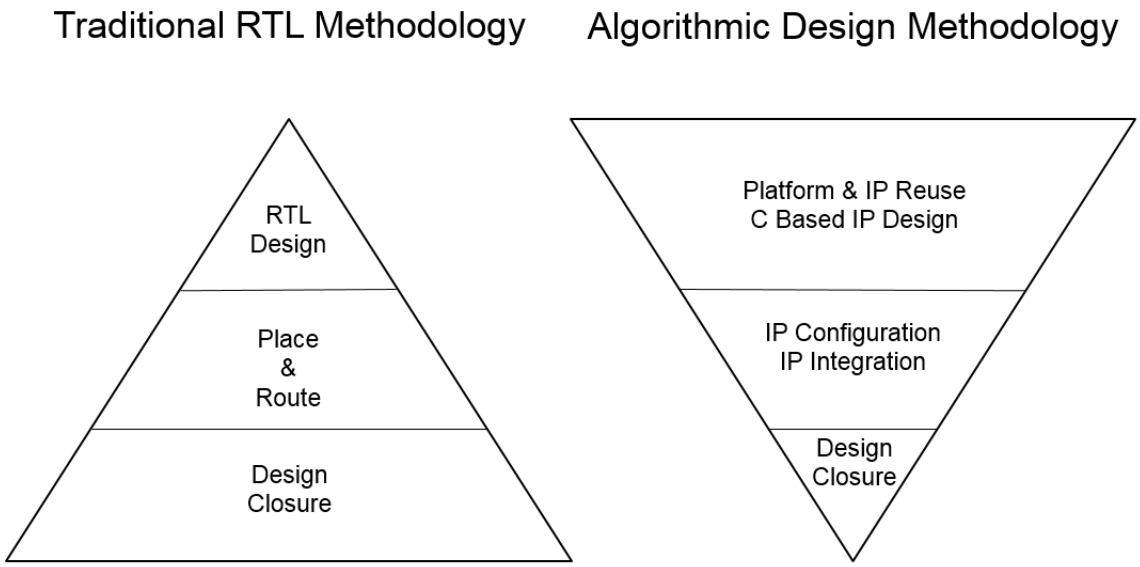

Figure 1-1: **High-Level Productivity Design Methodology Comparison**

Traditional Methodology

Traditional design development starts with experienced designers estimating how their design will be implemented in a new technology, capturing the design in Register Transfer Levels (RTLs), performing a few trials through synthesis and place and route to confirm their estimations and then proceeding to capture the remainder of the design. This is done while typically synthesizing each block in turn to re-confirm that the implementation details are acceptable.

The primary method for confirming that the design provides the intended functionality is to simulate the RTL. The detailed bit-accurate and cycle-accurate nature of an RTL description, although highly accurate, makes this process both slow and error prone.

Only when all blocks in the design have been captured in RTL can a full verification of the system be performed, often resulting in adjustments to the RTL. After all blocks in the system have been verified, they can be placed and routed together, and the accuracy of earlier estimations of timing and area can be either fully confirmed or shown to be inaccurate. This also often results in changes to the RTL, re-initiating another verification of the system and another re-implementation.

Designers are now often required to implement hundreds of thousands of lines of RTL code in a given project and spend much of their design time on implementation details. As highlighted in Figure 1-1, designers spend considerably more of their time implementing the design, rather than designing the novel and innovative solutions that all products require to remain competitive.

Moving to a newer technology to improve performance or a slower technology to provide more competitive pricing often means the majority of the RTL has to be re-written; designers must re-implement the amount of logic between the registers.

High-Level Productivity Design Methodology

The High-Level Productivity Design Methodology traverses the same basic steps as a more traditional RTL methodology, as shown in Figure 1-1. However, it allows designers to spend more time designing value-add solutions. The main attributes of a high productivity methodology are:

- The concept of a shell that is developed and verified in parallel with the differentiated logic. This shell encompasses the differentiated logic that captures the I/O peripherals and interfaces in a separate design project
- Using C-based IP simulation to decrease simulation times by orders of magnitude over traditional RTL simulation, providing designers the time to design the ideal solution.
- Using the Xilinx Vivado® Design Suite to highly automate the path to timing closure through the use of C-based IP development, IP re-use and standard interfaces.
    - Making use of the Vivado IP Catalog to easily re-use your own block and component level IP and provide easy access to the Xilinx IP already verified and known to implement well in the technology.

All steps in the High-Level Productivity Design Methodology can be performed interactively or using command line scripts. The result of all manual interactions can be saved to scripts, allowing the entire flow to be fully automated, from the design simulation through to programming the FPGA. Depending on your design and the runtime of the RTL system level simulation, this flow makes it possible to generate an FPGA bitstream and test the design on the board, often before any RTL design simulation has completed.

Even greater productivity improvements come when design derivatives are created. C-based IP is easily targeted to different devices, technologies, and clock speeds: as easy as changing a tool option. A fully scripted flow, with automated timing closure through C synthesis, means derivative designs can be quickly verified and assembled.

Design Process

The steps in the design process are shown in the following figure.

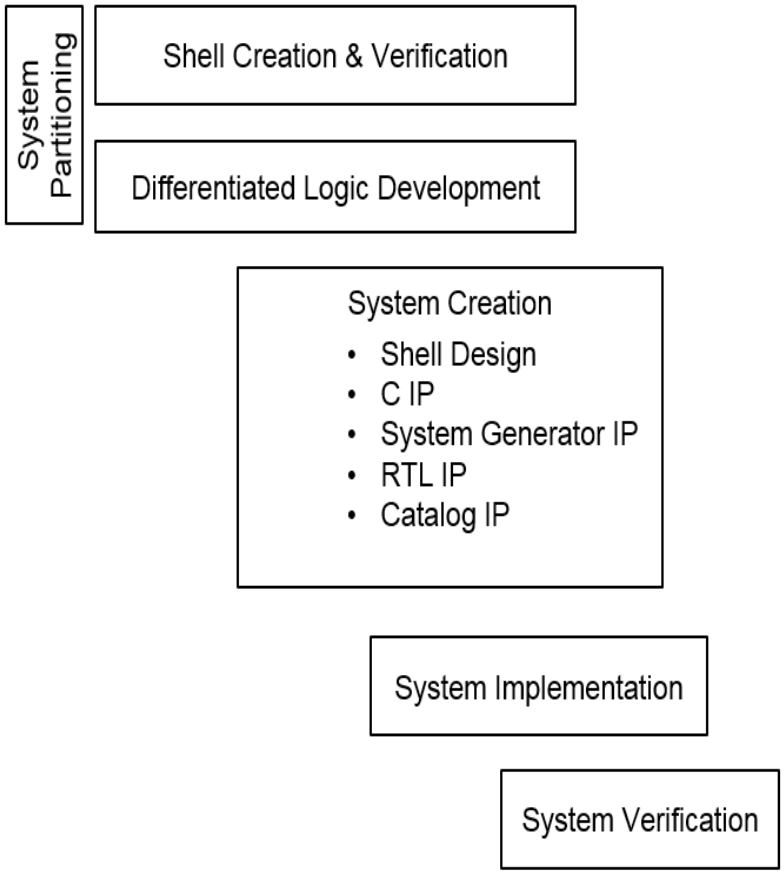

Figure 1-2: **High-Level Productivity Design Flow**

After the initial stage of system partitioning, described in Chapter 2, System Design, a key feature of this design flow is the overlapping nature of the development.

- **A Shell Development Flow**: Through the use of Vivado IP Integrator and the IP Catalog, the Vivado Design Suite enables fast, efficient block-level integration. Much of the critical aspects for system performance, including detail orientated interface creation, verification, and pin-planning, can be separated into a parallel development effort and given the focus they require. This flow is described in Chapter 3, Shell Development.
- **C Based IP Development**: It takes approximately 1 or 2 days to simulate a full frame of video using RTL simulation (depending on the design, the host machine, etc.). It takes approximately 10 seconds to perform the same bit-level accurate simulation using

C/C++. The productivity benefits of a C-based development flow cannot be ignored. This flow is described in Chapter 4, C-Based IP Development.

- **System Creation**: Vivado IP integrator and the IP catalog allow C-based IP to be quickly combined into a system block design using the shell design, legacy RTL IP, System Generator IP, and Xilinx IP. Automated interface connections and the ability to script the system creation mean that the system can be generated and re-generated quickly throughout the IP development process. This flow is described in Chapter 5, System Integration.
- **System Implementation**: You can ensure that minimal time is spent on design closure by using a shell design that is already verified, C-based IP automatically optimized for the device and clock frequency, and existing verified IP, all connected through industry standard Arm AMBA® AXI4 protocol-compliant interfaces. This flow is launched from the system block design with a few clicks of the mouse or using a scripted flow. This flow is described in Chapter 5, System Integration.
- **System Verification**: This is be performed using gate-level accurate RTL simulations and/or by programing the FPGA and verifying the design on the board. Because the RTL simulations are used to verify the system—not the iterative simulations used to validate the design during development—only a single simulation is required at the end of the design flow. This flow is described in Chapter 5, System Integration.

Accessing Documentation and Training

Access to the right information at the right time is critical for timely design closure and overall design success. Reference guides, user guides, tutorials, and videos get you up to speed as quickly as possible with the Vivado Design Suite. This section lists some of the sources for documentation and training.

Using the Documentation Navigator

The Vivado Design Suite ships with the Xilinx Documentation Navigator, as shown in Figure 1-3, which provides an environment to access and manage the entire set of Xilinx software and hardware documentation, training, and support materials. The Documentation Navigator allows you to view current and past Xilinx documentation. You can filter the documentation display based on release, document type, or design task. When coupled with a search capability, you can quickly find the right information. Methodology Guides appear as one of the filters under Document Types, which allows you to reach any of Methodology Guides almost instantaneously.

Xilinx uses the Documentation Navigator to provide you with up-to-date documentation using the Update Catalog feature. This feature alerts you about available catalog updates and provides details about the documents that are involved. Xilinx recommends that you

always update the catalog when alerted to keep it current. Additionally, you can establish and manage local documentation catalogs with specified documents.

The Documentation Navigator has a tab called the **Design Hub View**. Design hubs are collections of documentation related by design activity, such as Applying Design Constraints, Synthesis, Implementation, and Programming and Debug. Documents and videos are organized in each hub in order to simplify the learning curve for that area. Each hub contains sections such as Getting Started, Support Resources (with an FAQ for that flow), and Additional Learning Materials. For new users, the Getting Started section provides a good place to start. For those already familiar with the flow, Key Concepts and the FAQ may be of particular interest to gain expertise with the Vivado Design Suite.

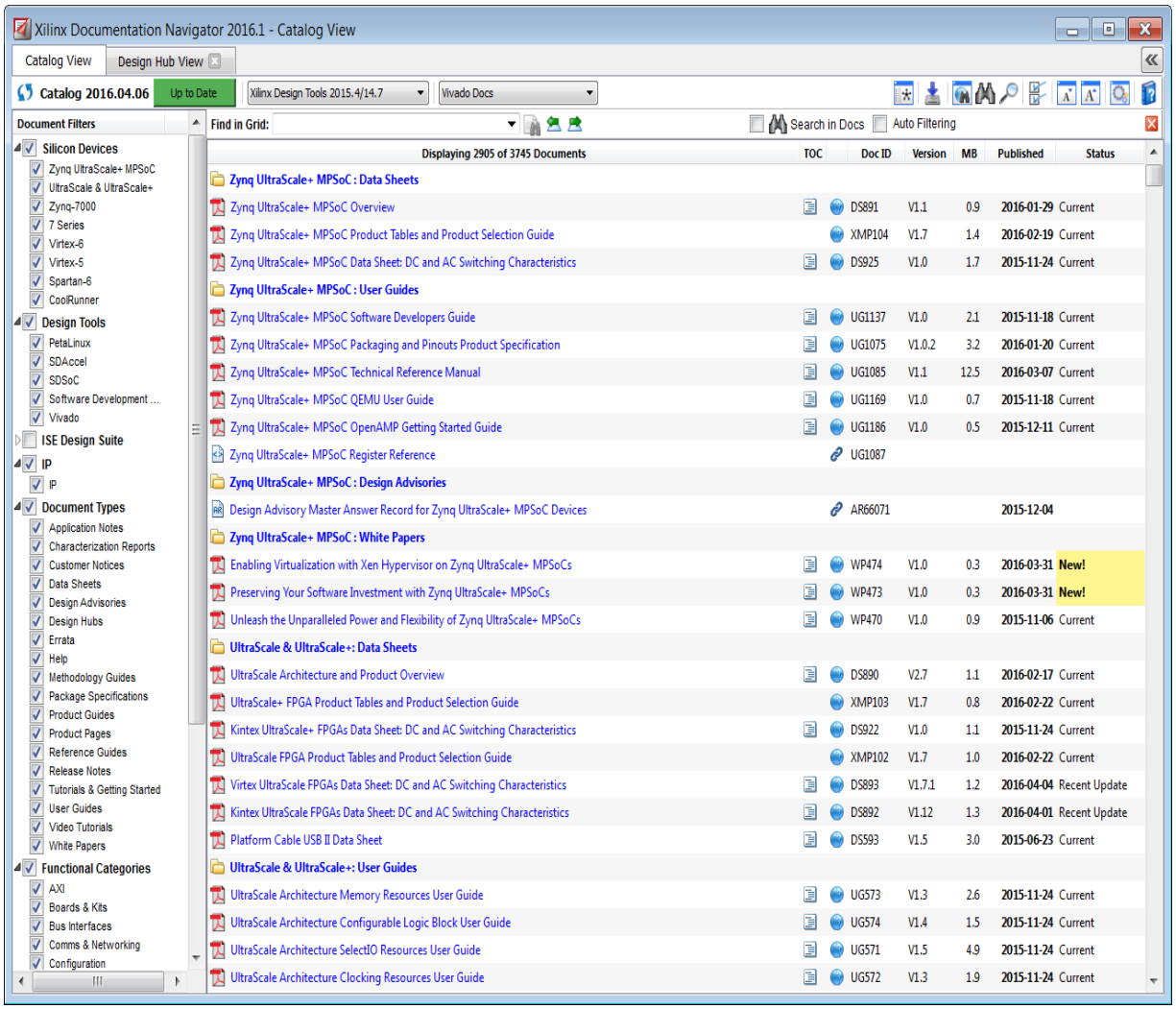

Figure 1-3: **Xilinx Documentation Navigator**

Chapter 2

System Design

Overview

Before starting on your project, it is important to have a clear understanding of how you will design and assemble your system. In any complex system, there are multiple paths to a solution. These paths are dictated by the various choices you make on what IP blocks to create from scratch, what IP you can re-use, and the tools and methodology used to validate the IP, integrate the IP in a system, and verify the system.

This chapter addresses the system partitioning choices you will make and reviews key features of the Vivado® Design Suite which help automate the process of system development.

- System Partitioning
- System Development

System Partitioning

In a typical design, the logic on the periphery of the design is dedicated to interfacing with external devices, typically using standard interfaces. Example of this are DDR, Gigabit Ethernet, PCIe, HDMI, ADC/DAC, and Aurora interfaces. These interfaces and the components used to implement them are typically standard to multiple FPGA designs within the same company.

In the High-Level Productivity Design Methodology this logic is separated from the differentiated logic and is considered the shell. The figure below shows an example shell block design. The shaded area in the center of the figure below indicates where the differentiated logic or shell verification IP can be added.

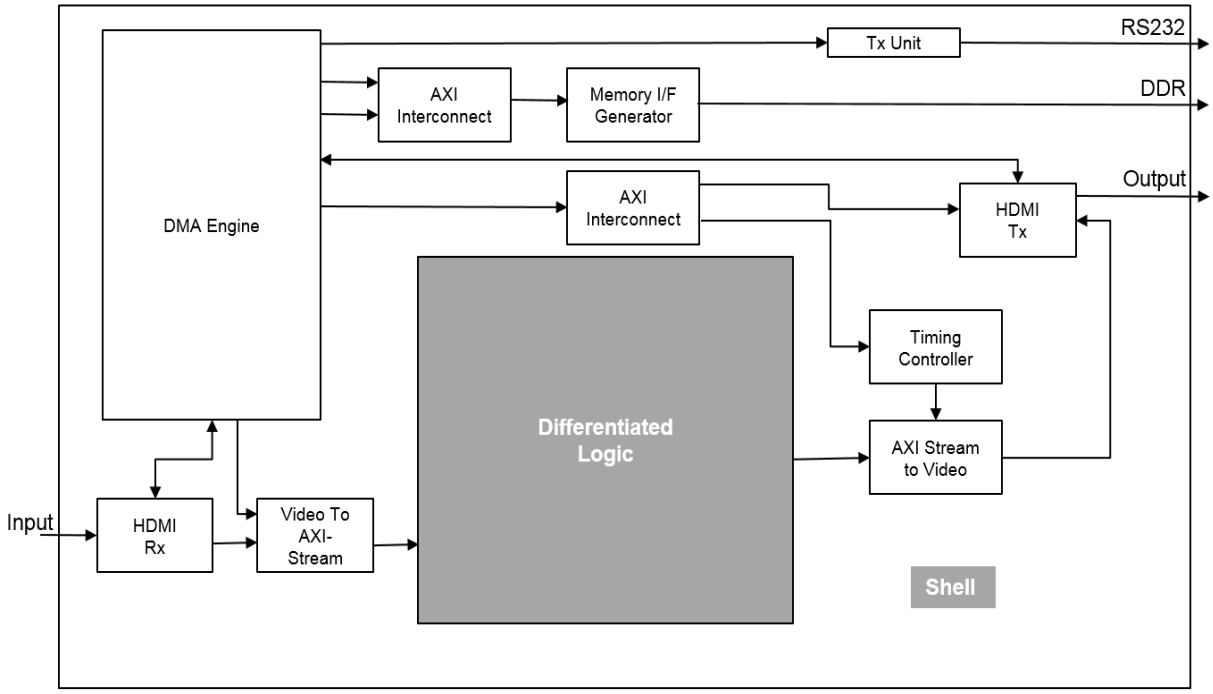

Figure 2-1: **Shell Design Example**

Key benefits of this methodology are:

- The shell is developed and verified independently of the rest of the design.
- Board-level integration and device pin planning are addressed by a separate dedicated team working in parallel.
- The shell is saved and re-used (even re-edited), allowing multiple derivative designs to be quickly realized.
- The differentiated logic is developed and verified independently of the shell.
- The pre-verified shell and differentiated logic are quickly integrated into a complete system.

When partitioning your system, the first task is to determine what will be implemented in the shell and what will be implemented as differentiated logic.

Shell Design

A shell design provides two key attributes to a high productivity methodology:

- Separating standard interface logic from the differentiated logic, allowing the development and verification of both to proceed in parallel.

- Creating a re-usable design, or shell, that can be used to quickly create design derivatives. A shell should ideally contain the parts of the design that are standard, such as design interfaces and interface IP. However, a shell can also contain blocks used for pre-processing or post-processing. If the processing functions are independent of the core design IP, and if the processing functions could be used across multiple designs, it is more ideal to place these blocks in the shell. The shell re-use methodology allows blocks to be easily removed from the shell.

Irrespective of the logic you decide to incorporate into the shell design, a key attribute of the shell design is that the internal interfaces, those which connect to the internal design IP, should be implemented using standard interfaces. Use of standard internal interfaces such as AXI will enhance the shell re-use by doing the following:

- Allowing the shell to be easily connected to a design IP that has yet to be developed
- Ensuring that verification of the shell also verifies the internal interfaces
- Enabling use of the high productivity integration features described in IP Integrator and Standard Interfaces

Even if you are initially only thinking of one design, a shell based methodology allows you to easily create derivative designs after the initial design is implemented.

More details about shell development and verification are described in Chapter 3, Shell Development.

IP Design

The key feature of the IP development flow is that it includes only the IP that differentiates the product from the shell.

The design IP is not standard and will be developed. Much of the development effort is running simulations to validate that the design provides the correct functionality. This effort can be minimized, and simulation run times improved, by not including standard blocks that do not impact the new functionality being developed; those blocks should be in the shell.

The figure below shows a representation of a complete system with design IP added to the shell design. A key feature of the completed system is that it might contain IP developed from many different sources, such as:

- IP generated from C/C++ using Vivado HLS
- IP generated from System Generator
- IP from RTL
- Xilinx® IP
- Third-Party IP

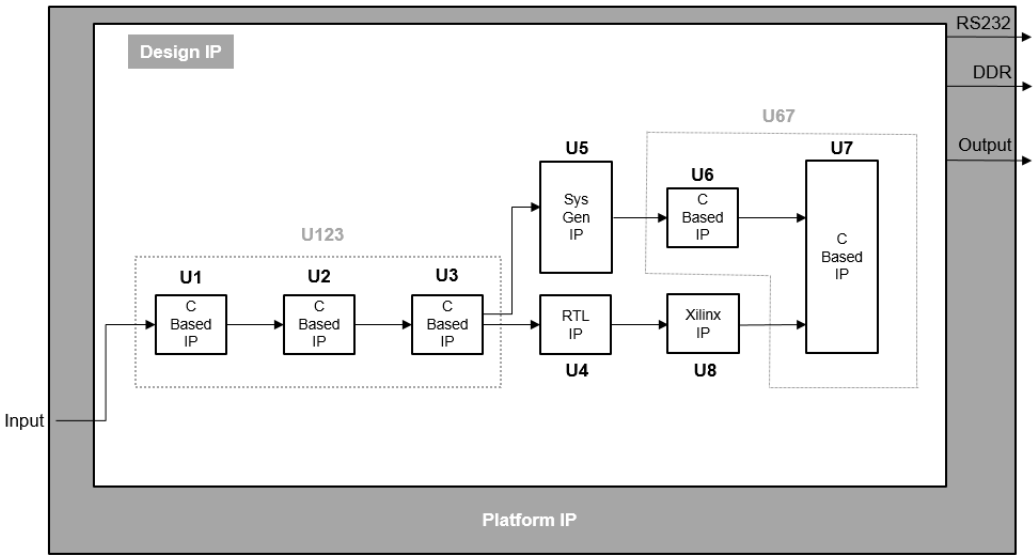

Figure 2-2: **System Design Example**

In a high productivity design methodology, one of the greatest benefits comes from the verification speed of C simulation. From a design creation perspective, there is large productivity gain by simulating the C blocks together during development.

- Fast C simulation allows the designer to quickly develop and verify an accurate solution.
- Multiple C blocks simulated together help each verify the output of the other.
- A larger overall productivity benefit can be achieved if several C IP are combined together in a C simulation.

The figure above, Figure 2-2, highlights a dilemma you may face when using C IP. Blocks U1, U2, and U3 are all C IP and could be grouped into a single top-level U123. Similarly, blocks U6 and U7 could be grouped into a single IP block, U67. You can do one of the following:

- Create multiple smaller C IP blocks, such as U1, U2, U3, U6 and U7.
- Create a few large C IP blocks, such as U123 and U67 outlined in the figure above.

From a design integration perspective, there is no difference between these methods; if the IP blocks are generated with AXI interfaces, they are easily integrated together using IP integrator. For designers who are new to C-based IP development, it might make more sense to work on smaller blocks, learn how to optimize each small block independently, and then integrate multiple smaller IPs together. For designers who are comfortable with C IP development, it might make more sense to generate a few large C IP blocks.

IMPORTANT: *The key productivity benefit is being able to simulate as many C IP blocks as one C simulation during development.*

In the situation described above, the same C test bench that verifies blocks U1, U2, and U3 would be used to verify U123. The difference in IP generation is that you either set the top level for C synthesis in Vivado HLS as function U123, or as function U1 followed by U2 and U3.

Regardless of which route is taken to create the IP blocks, each of the IP blocks should be verified in isolation as follows:

- IP developed from C/C++ is verified using the C/RTL co-simulation feature of Vivado HLS, allowing the RTL to be verified using the same C test bench used to verify the C based IP.
- IP developed from System Generator is verified using the MathWorks Simulink design environment provided in System Generator. The Simulink environment enables the easy generation of complex input stimuli and analysis of complex results through the use of pre-defined simulation elements. IP generated from C/C++ and through traditional RTL can be imported into the System Generator environment to take advantage of this verification.
- For IP generated from RTL, you must create an RTL test bench to verify the IP.
- IP provided by Xilinx and third-party providers is pre-verified, however you might wish to create a test bench to confirm its operation based on your own set of configuration parameters.

The use of standard AXI interfaces on the IP allows the IP to be quickly integrated, both with each other and with the shell design.

System Development

Although the concept of using a shell and multiple IP blocks is not new to FGPA designers, this methodology typically requires lots of RTL to be developed and simulated, requiring stitching hundreds, if not thousands, of individual RTL signals together multiple times to make the following connections:

- The shell to verification IP
- The shell to the core design IP
- The shell to derivative core design IP.

In lieu of the many additional man-hours in both design and verification effort it would take to use this methodology in a traditional RTL design flow (this is an error prone task when performed in a text editor), design teams typically design and integrate everything together.

Vivado IP integrator enables this methodology and allows IPs to be quickly integrated without the traditional hand-editing of RTL files.

The key features for using this methodology are:

- Vivado IP Catalog
- IP integrator and standard interfaces

Vivado IP Catalog

The Vivado IP Catalog is the backbone of any methodology that uses IP and IP re-use. Figure 2-3 shows an alternative view of the design process for the High-Level Productivity Design Methodology highlighting where and when the IP Catalog is used.

IMPORTANT: *Use of the IP Catalog is key to enabling a High-Level Productivity Design Methodology.*

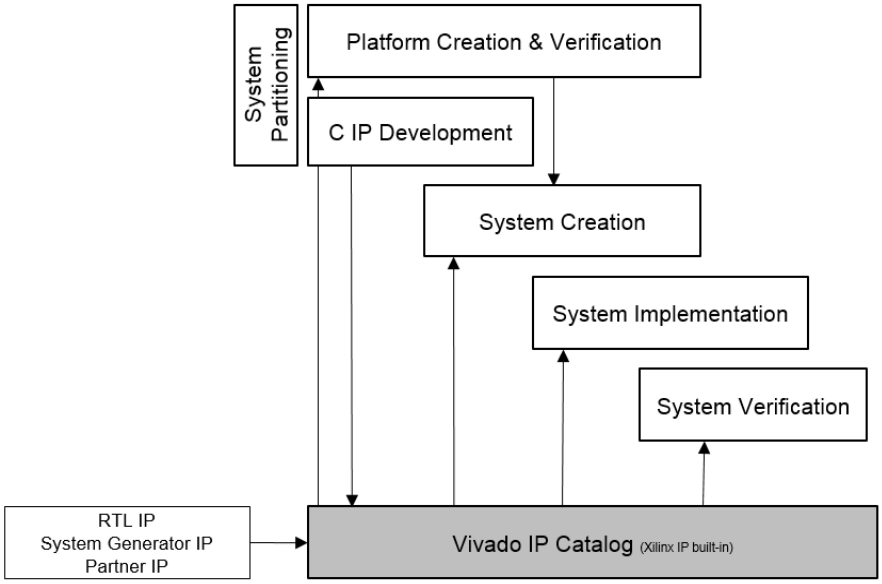

Figure 2-3: **IP Catalog and the Design Process**

The IP Catalog has the following features:

- Includes approximately 200 IPs from Xilinx. For more information, refer to the Xilinx Intellectual Property Page [Ref 12].
- Saves the output from C based IP development.
- Can be enhanced with System Generator, legacy RTL, and Xilinx Partner IP.
- Contains a large number of interface IPs, supports the use of legacy RTL IP, and is used extensively when creating the shell.
- Is the source for all IP blocks during system integration.

- Provides the RTL implementations used during system integration and verification.

During shell development, the shell is assembled in IP integrator using IP from the IP Catalog. This might include interface IP provided by Xilinx (Ethernet, VGA, CPRI, Serial Transceivers, etc.), IP from Xilinx partners, legacy RTL package as IP for the IP Catalog, or IP created by Vivado HLS and System Generator.

Details about packaging legacy RTL as IP are provided in the *Vivado Design Suite Tutorial: Creating and Packaging Custom IP* (UG1119) [Ref 5].

Details on creating IP with AXI interfaces from System Generator are provided in the *Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator* (UG897) [Ref 6].

The default output from Vivado HLS is an IP packaged for the IP Catalog. This is described in IP Packaging.

IP Integrator and Standard Interfaces

The Vivado IP integrator allows IP blocks to be quickly added to a canvas and connected, and is a key enabler of a high productivity design methodology.

IMPORTANT: *The key to high productivity using Vivado IP integrator is the use of standard interfaces.*

Figure 2-4 shows an example block design captured in IP integrator.

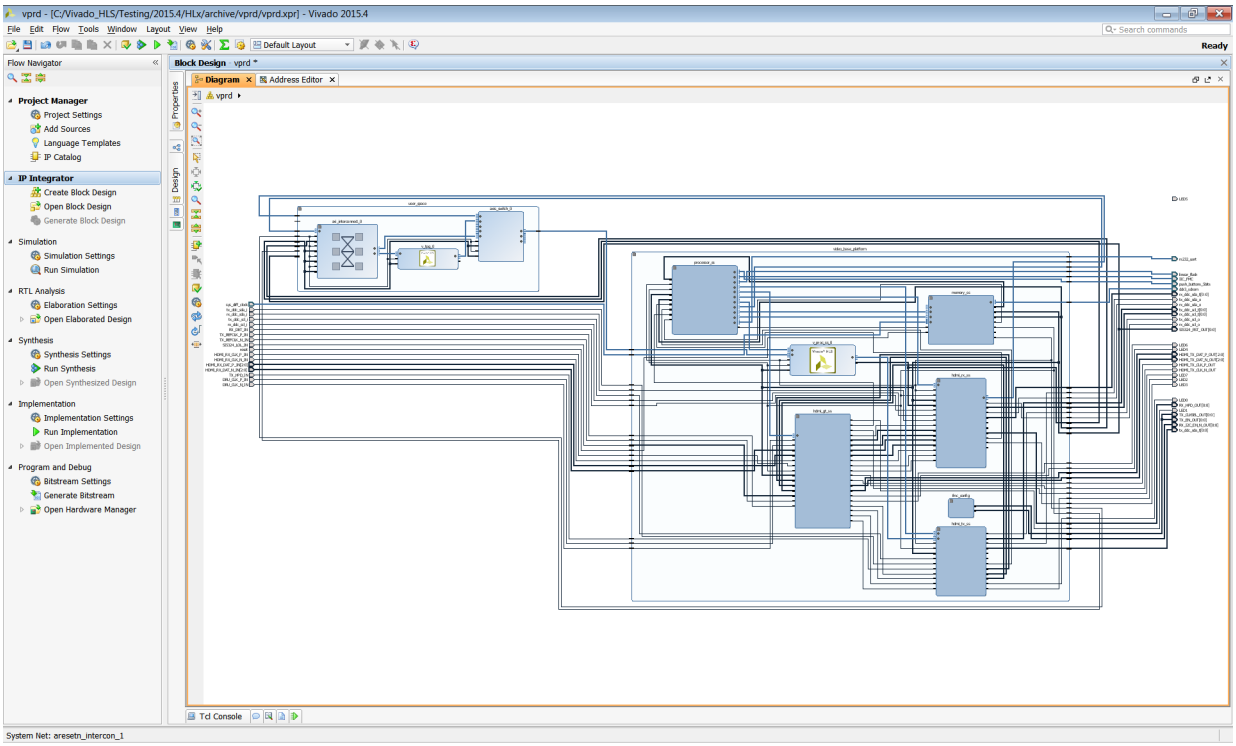

Figure 2-4: **IP integrator block design**

The connection types include:

- Pin-level connections such as clock and reset signals.
- Bus-level connections such as AXI, AXI4-Lite and AXI4-Stream buses.
- Board-level connections such as DDR.

Connections in IP integrator are made by using the mouse to graphically connect the pins on each IP together. In addition to supporting basic connecting bit-level connections, support is provided for bus-level connection and designer assistance.

The following figure highlights the advantage of bus-level connections. In this example, two AXI master interfaces are to be connected together. Note that as soon as the connection is made to the first port, all possible valid connections are identified on the diagram by green check marks.

IMPORTANT: *IP integrator does not allow illegal connections to be made. This eliminates the types of connectivity errors typically made when this process is performed through manual edits.*

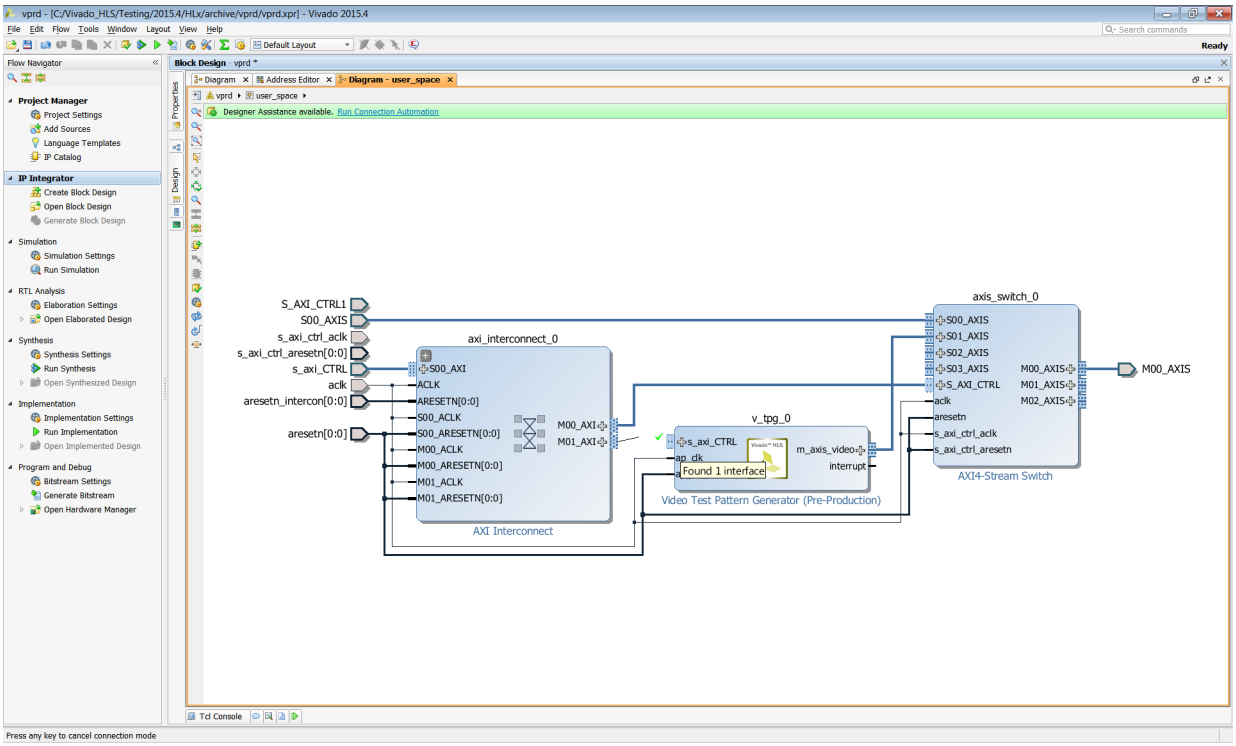

Figure 2-5: **Connection Automation**

A further productivity feature provided by the use of standard AXI interfaces and IP integrator is the automatic generation of the AXI interconnect IP. Figure 2-6 shows the result when connecting:

- An AXI output on one block
- An AXI4-Stream input on another block

IP integrator automatically adds the AXI Interconnect IP to connect the master type interface to a stream type interface.

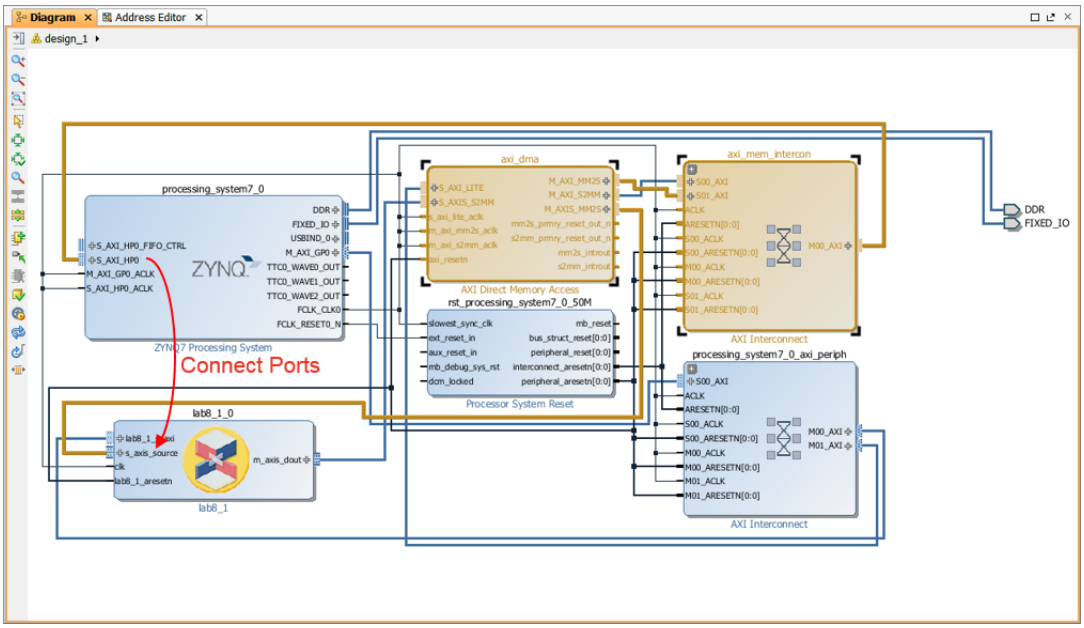

Figure 2-6: **Automated AXI Interconnect IP**

This AXI Interconnect IP is provided in the IP Catalog and can be added manually but IP integrator automates the task. Furthermore, if the final block design is saved as a script, the Tcl commands simply state which pins to connect.

TIP: *When you upgrade to a new release of the Vivado Design Suite and Xilinx IP, re-run the script to make sure that it uses the latest interconnect logic.*

A final case for using standard interfaces in your designs is the designer assistance provided for board level connections. In addition to allowing the target device to be selected, the Vivado Design Suite allows selection of target boards. IP integrator is board-aware and is able to automate board-level connections.

After confirmation from the designer, IP integrator automatically makes the connections between the IP and FPGA pins (board connections).

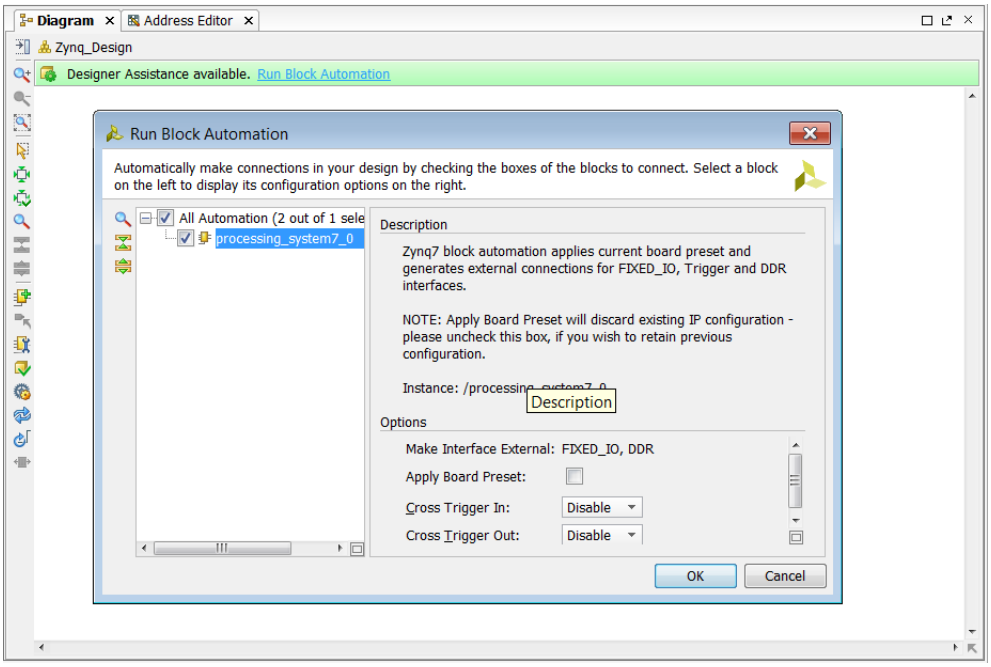

Figure 2-7: **Block Automation**

IP integrator automates integration of IP into a block level diagram. Additional features include Design Rule Checks using the Validate feature and the automatic addition of clock and reset logic for AXI Interconnect IP. The key to taking advantage of this automation, and enabling a productive shell methodology, is the use of standard interfaces and AXI interfaces for on-chip communication.

Chapter 3

Shell Development

Overview

The use of a shell is a critical part of the productivity benefits provided by a high productivity design methodology. A shell design contains all of the standard interfaces and processing blocks that provide the connection between the core design IP and the remainder of the system, and is developed in parallel to the core design IP.

A methodology that incorporates shell design provides key productivity gains, such as:

- It allows the development of the design interfaces and I/O planning to proceed independently of the core design.
- It enables the verification of the design interfaces to start before the core design IP is ready.
- It reduces the verification time for the interfaces because the design is smaller; it does not contain the core design IP that typically represents the majority of the logic in the system.
- It promotes a highly productive design re-use methodology, allowing derivative designs to be easily created.

An overview of the shell design methodology is shown in Figure 3-1. A key attribute of the methodology is the re-use of the shell design.

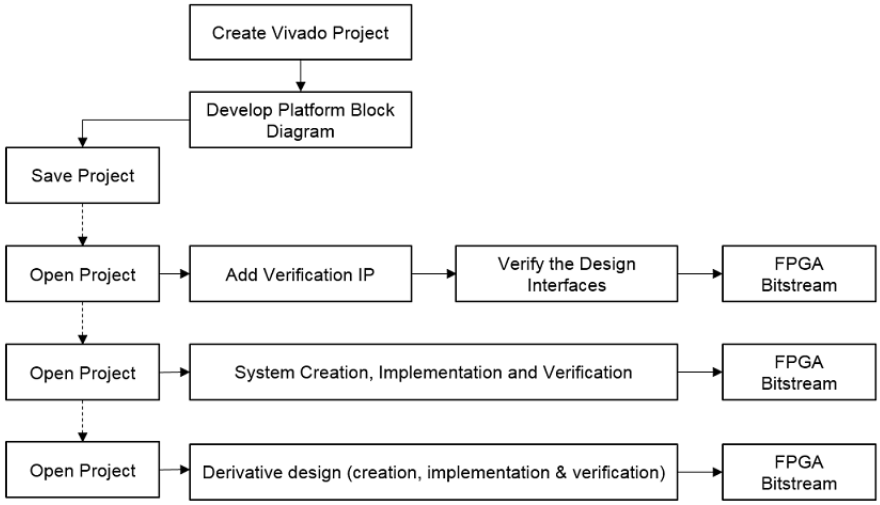

Figure 3-1: **Shell Methodology**

Shell development consists of two equally important processes: shell design and shell verification.

Shell Design

A shell design consists only of the design periphery, as shown in the figure above, and must be in a form that makes the design easy to re-use. The shell will be saved and re-opened to form the basis of multiple projects.

To achieve the level of design re-use required to enable the flow shown in the above figure, the shell design should be captured as a block design in IP integrator that can be easily saved and re-opened to form the basis of other design projects.

Assemble Existing IP

A shell design is assembled as block design in IP integrator using IP from the IP Catalog.

IMPORTANT: *In preparation for creating your shell, package any existing RTL or company-specific IP you want to use in the shell design as IP for use in the IP Catalog. This allows you to add the IP in the shell block design.*

For details about how to package blocks for the IP Catalog, refer to the *Vivado® Design Suite Tutorial: Creating and Packaging Custom IP* (UG1119) [Ref 5].

Shell Design Project

After you assemble your IP, create a Vivado RTL project.

TRAINING: *Creating a Vivado RTL project is detailed in the* Vivado Design Suite QuickTake Video: Creating Different Types of Projects.

When creating the Vivado project:

- Specify the project as an RTL project and select **Do not specify any sources at this time**. The sources for the shell design are the IP you have packaged in the IP Catalog.
- Ideally, select the target as a Xilinx® board. The I/O for devices used on Xilinx boards has already been configured. This allows you to get started in the quickest possible time—while your own custom board is developed—and allows you to use Designer Automation within IP integrator for I/O connections.

If you do not specify a Xilinx board as the target, you will also need to specify the I/O connections for the target device. Refer to this link in the *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949) [Ref 7].

If you are using your own custom board during the development process, you might want to create a board file that details the board connections and allows designer automation within IP integrator, greatly simplifying board level connections. Details on the board file are provided in "Using the Vivado Design Suite Board Flow," provided at this link in the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895) [Ref 9].

After the project has been created, use the **Create Block Design** button in the Flow Navigator to open IP integrator and create a new Block Design. In the IP integrator window, specify the source of your IP repository and use the **Add IP** button to start assembling the shell.

Once the shell is complete, the `write_bd_tcl` command is used to save the entire block design as a Tcl script. This script contains everything that is needed to re-generate the block design from scratch. The block design and Vivado project are saved and ready for the next stages of verification and system development.

In depth information on pin planning, IP integrator, and other features in the Vivado Design Suite is provided in the Design Hubs tab of the Documentation Navigator. For more information, refer to Using the Documentation Navigator.

Shell Verification

After the shell design is created, shell verification can proceed. In the verification process, the shell design is re-opened and verification IP is added to the design to confirm that the interfaces are working.

Shell Verification Projects

The first step in verifying the shell design is to create a new verification project using one of the following two options.

- Open the Vivado project for the shell design and use **File > Save Project As** to save the shell design in a new project.
- Create a new Vivado RTL project (with no RTL sources) and the same target device or board. Then select **Create Block Design**, and in the console, source the Tcl script saved using `write_bd_tcl` to regenerate the shell Block Design in the new project.

Multiple verification projects might be required to ensure that the complexity of the verification design is manageable. The figure below shows an example shell verification design. This example tests only a single interface.

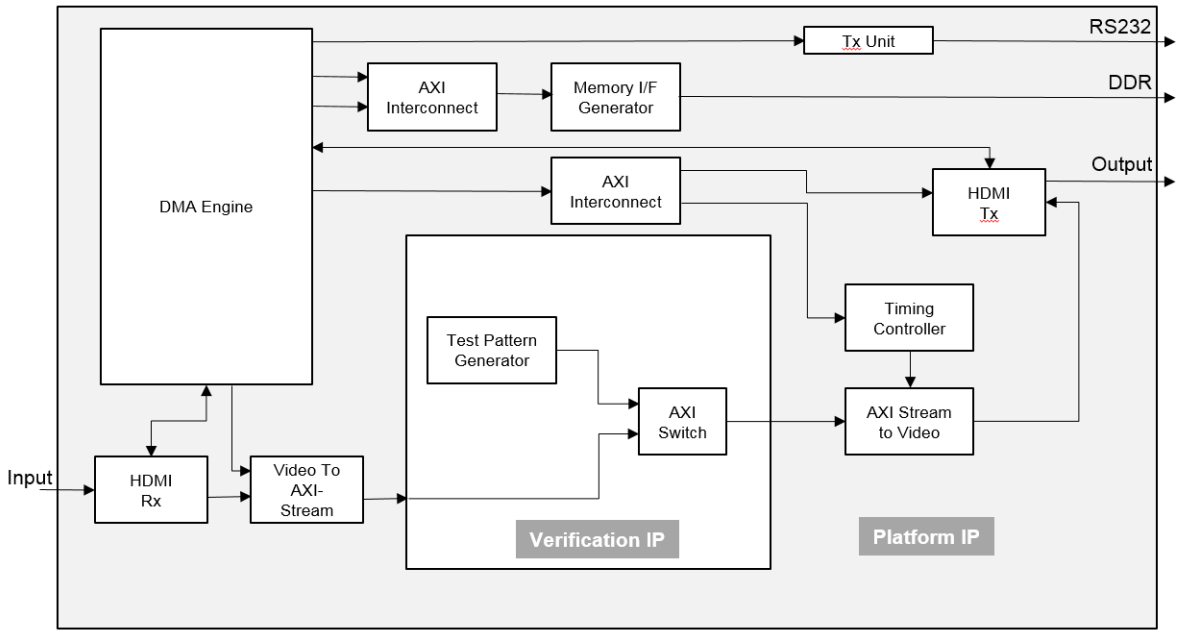

Figure 3-2: **Shell Verification Example**

Verification IP

Verification IP is added to the shell design from the Vivado IP Catalog to verify the design.

The verification IP can be developed using any of the techniques discussed in this guide: RTL, System Generator, or C-based IP. The following examples shows how if standard AXI Interface IPs are used, a small C file can be used to quickly create, for example, a HANN window of N samples output on an AXI4-Stream interface. An AXI memory mapped interface can be implemented by simply changing the interface directive to `m_axi` from `axis`, as shown in the following code example:

```
void verify_IP_Hann(float outdata[WIN_LEN]) {
// Specify AXI4-Stream output
#pragma HLS INTERFACE axis port=outdata
// Alternative output AXI4M (commented out)
//#pragma HLS INTERFACE m_axi port=outdata

    float coeff[WIN_LEN];
    coeff_loop:for (int i = 0; i < WIN_LEN; i++) {
       coeff[i] = 0.5 * (1.0 - cos(2.0 * M_PI * i / WIN_LEN));
    }

    winfn_loop:for (unsigned i = 0; i < WIN_LEN; i++) {
      outdata[i] = coeff[i];
      }
}
```

For information about how to use Vivado HLS to create interface blocks between other IPs, refer to *Methods for Integrating AXI4-based IP Using Vivado IP Integrator Application Note* (XAPP1204) [Ref 10].

Verifying the Shell

If a top-level test bench is added to the simulation sources, the shell design can be verified by simulation prior to programming the FPGA.

Verification of the shell using RTL simulation requires the creation of the RTL test bench. This same test bench is used to verify the fully integrated design. If multiple verification projects are used to verify the shell, the same test bench should be expanded to verify all of the interfaces.

To verify detailed interfaces on the FPGA, additional signal level debug probes can be added to the design.

When working in the Block Design, the right-click menu allows nets to be easily marked for debug. Signals marked for debug can be analyzed during hardware operation: ILA cores are added to the design to allow the signals to be captured and scanned out of the FPGA for analysis. Refer to this link in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 8].

The final design is then processed through the Vivado design flow to bitstream. When the shell is fully verified, any modifications to the shell design, other than modifications to the verification IP, should be propagated back to the original source shell design project. The shell design is then ready for the integration of the core design IP.

Chapter 4

C-Based IP Development

Overview

In a high productivity design flow, the primary means of generating the core design IP is through the use of C-based IP and High-Level Synthesis (HLS) of C code into RTL. A C-based IP development flow provides the following benefits:

- Superior simulation speeds provided by C verification
- Automated generation of accurately timed optimized RTL
- Ability to use existing C IP from libraries
- Ease of integrating the resulting RTL IP into a complete system using IP Integrator

This chapter discusses how C-Based IP is created, validated, synthesized, analyzed, optimized, and packaged into IP for the IP Catalog. The means for achieving this is Vivado® High-Level Synthesis (HLS), a tool provided as part of the Vivado Design Suite.

The design flow for Vivado HLS is shown in the figure below. The design flow steps are:

1. Compile, execute (simulate), and debug the C algorithm.

   Note: In high-level synthesis, running the compiled C program is referred to as *C simulation*. Executing the C program simulates the function to validate that the algorithm is functionally correct.

2. Synthesize the C program into an RTL implementation, optionally using user optimization directives.
3. Generate comprehensive reports and analyze the design.
4. Verify the RTL implementation using a push button flow.
5. Package the RTL implementation into a selection of IP formats.

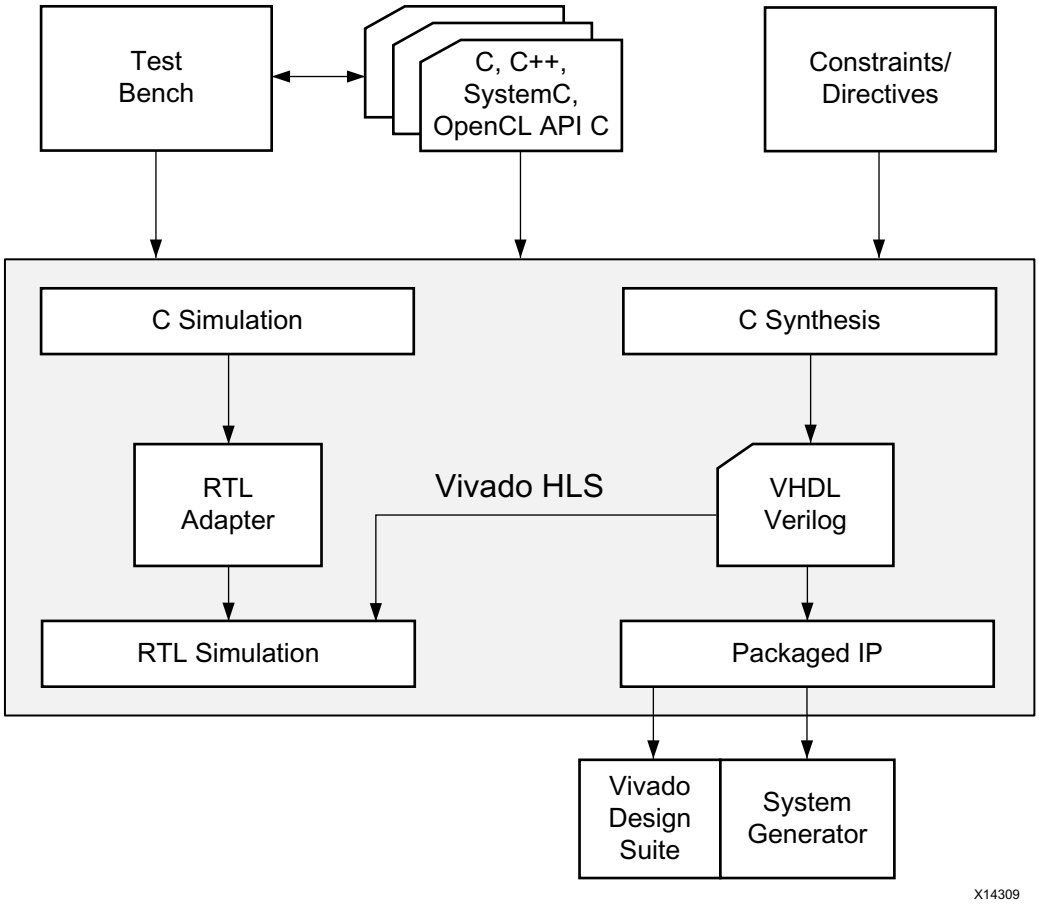

Figure 4-1: **Vivado HLS Design Flow**

Details about using Vivado HLS are provided in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2]. This chapter explains a methodology for using Vivado HLS in a highly productive manner.

Fast C Verification

Simulating an algorithm in C can be orders of magnitude faster than simulating the same algorithm in RTL.

For example, consider a standard video algorithm. A typical video algorithm in C processes a complete frame of video data and compares the output image against a reference image to confirm that the results are correct. The C simulation for this typically takes 10-20 seconds. A simulation of the RTL implementation typically takes a few hours to day(s) depending on the number of frames and the complexity of the design.

The more development performed at the C level, using the simulation speed of software, the more productive you will be. It is at this level that designers actually design: adjusting their algorithm, data types, and bit-width to validate and confirm that the design is correct.

The remainder of the flow is development: using a tool chain to implement the correct design in an FPGA. The benefits provided by the Vivado Design Suite and a High-Level Productivity Design methodology is the high degree of automation provided to the development flow.

After the initial FPGA design is implemented, it is not uncommon to create an entire new bitstream to program the FPGA, using the scripted flow presented in Chapter 5, System Integration, in less time than it takes to perform a system-wide RTL simulation.

To maximize the productivity of a C-based IP flow, the following should be understood:

- C Test Bench
- Self-Checking Test Bench
- Bit Accurate Data Types

C Test Bench

The top level of every C program is the `main()` function. Vivado HLS synthesizes any single function below the level of `main()`. The function to be synthesized by Vivado HLS is referred to as the Design Function. This is highlighted in Figure 4-2.

- All functions below the Design Function are synthesized by Vivado HLS.
- Everything outside of the Design Function hierarchy is referred to as the C test bench.

The C test bench includes all of the C code below `main()` that supplies input data to the Design Function and accepts output data from the Design Function to confirm that it is accurate.

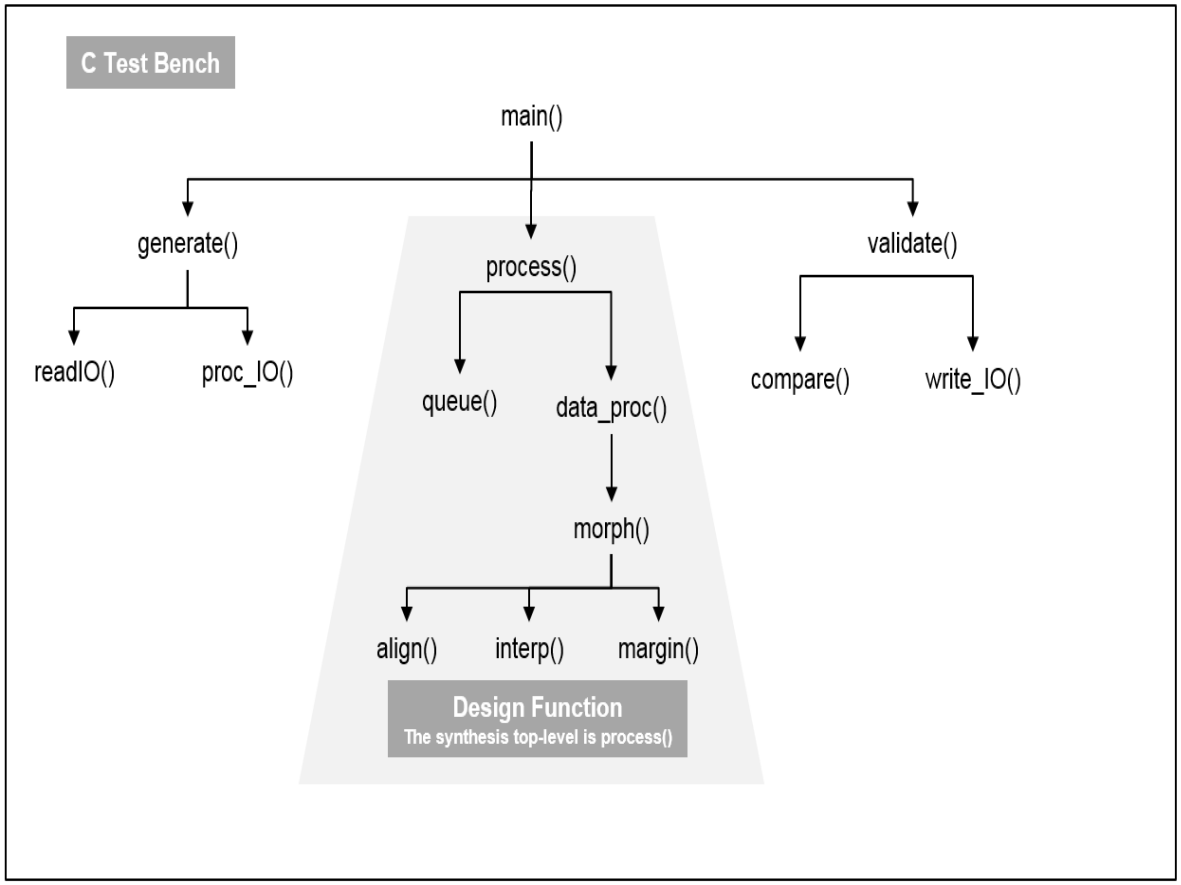

Figure 4-2: **C Test Bench**

The single biggest mistake made by users new to the Vivado HLS design flow is to proceed to synthesize their C code without using a C test bench and performing C simulation. This can be highlighted by the following code. What is wrong with this example of nested loops?

```
#include "Nested_Loops.h"

void Nested_Loops(din_t A[N], dout_t B[N]) {

  int i,j;
  dint_t acc;

  LOOP_I:for(i=0; i < 20; i++){
    LOOP_J: for(j=0; j < 20; j++){
      if(j=0) acc = 0;
        acc += A[i] * j;
      if(j=19) B[i] = acc / 20;
    }
  }
}
```

This code fails to synthesize into the expected result because the conditional statements evaluate as `FALSE` and J is set to `19` at the end of the first iteration of `LOOP_J`. The conditional statements should be `j==0` and `j==19` (using `==` instead of `=`). The preceding code example compiles, executes, and can be synthesized without any issue. However, it will not do what is expected by a cursory visual evaluation of the code.

In an era where developers consistently use one or more of C/C++, Perl, Tcl, Python, Verilog, and VHDL on a daily basis, it is hard to catch such trivial mistakes, more difficult still to catch functional mistakes, and extremely difficult and time consuming to uncover either after synthesis.

A C test bench is nothing more than a program that calls the C function to be synthesized, provides it test data, and tests the correctness of its output; this can be complied and run prior to synthesis and the expected results validated before synthesis.

You might initially feel that you are saving time by going directly to synthesis, but the benefits of using a C test bench in your design methodology are worth a lot more than the time it takes to create one.

Self-Checking Test Bench

The Vivado HLS supports C simulation prior to synthesis to validate the C algorithm and C/RTL co-simulation after synthesis to verify the RTL implementation. In both cases, Vivado HLS uses the `return` value of function `main()` to confirm that the results are correct. An ideal C test bench has the result checking attributes shown in the code example below. The outputs from the function for synthesis are saved into the file `results.dat` and compared to the correct and expected results, which are referred to as the “golden” results in this example.

```
int main () {
  ...
  int retval=0;
  fp=fopen("result.dat","w");
  ...
  // Call the function for synthesis
  loop_perfect(A,B);
```

```
  // Save the output results
  for(i=0; i<N;++i) {
    fprintf(fp, "%d \n", B[i]);
  }...

  // Compare the results file with the golden results
  retval = system("diff --brief -w result.dat result.golden.dat");
  if (retval != 0) {
    printf("Test failed  !!!\n");
    retval=1;
  else {
    printf("Test passed !\n");
  }

  // Return 0 ONLY if the results are correct
  return retval;
}
```

In the Vivado HLS design flow, the `return` value to function `main()` indicates the following:

- A value of Zero: Results are correct.
- A Non-zero value: Results are incorrect.

RECOMMENDED: *Because the system environment (for example, Linux, Windows, or Tcl) interprets the return value of the `main()` function, Xilinx® recommends that you constrain the return value to an 8-bit range for portability and safety.*

By using a self-checking test bench, you are not required to create an RTL test bench to verify that the output from Vivado HLS is correct. The same test bench used for the C simulation is automatically used during C/RTL co-simulation and the post-synthesis results verified by the test bench.

There are many ways in C to check that the results are valid. In the above example, the output from the function for synthesis is saved to file `result.dat`.and compared to a file with the expected results. The results could also be compared to an identical function not marked for synthesis (which executes in software when the test bench runs) or compared to values calculated by the test bench.

IMPORTANT: *If there is no `return` statement in function `main()` of the test bench, the C standard dictates that the `return` value is zero. Thus, C and C/RTL co-simulation always reports the simulation as passing, even if the results are incorrect. Check the results and return zero only if they are correct.*

The time spent to create a self-checking test bench ensures that there are no obvious errors in the C code and no requirement to create RTL test benches to verify that the output from synthesis is correct.

Bit Accurate Data Types

Arbitrary precision data types are provided with Vivado HLS and allow variables to be specified at any width. For example, variables might be defined as 12-bit, 22-bit, or 34-bits wide. Using standard C data types, these variables are required to be 16-bit, 32-bit, and 64-bit, respectively. Using the standard C data types often results in unnecessary hardware to implement the required accuracy; for example, 64-bit hardware when only 34-bit is required.

An even greater benefit of using arbitrary precision data types is that the C algorithm can be simulated using these new bit-widths and the bit-accurate results analyzed. For example, you might wish to design a filter with 10-bit inputs and 14-bit output and you might determine that the design can use a 24-bit accumulator. Performing C simulation—which can simulate the filter with tens of thousands of samples in a matter of minutes—will quickly confirm whether the signal-to-noise ratio of the output is acceptable. You will quickly be able to determine if the accumulator is too small or verify that using a smaller, more efficient accumulator still provides the required accuracy.

IMPORTANT: *Simulation of a bit-accurate C is the fastest way to verify your design.*

A productive methodology is to start your initial design with standard C data types and confirm the algorithm performs as designed. Then migrate your C code to use arbitrary precision data types. This migration to more hardware efficient data types can only be performed safely and productively if there is a C test bench checking the results, allowing you to quickly verify the smaller more efficient data types are adequate. Once you are comfortable with arbitrary precision types, you will typically use arbitrary precision data types right from the start of your new C project.

The benefits of using a C test bench, and the loss of productivity in not using one as part of your design methodology, cannot be overstated.

The Vivado HLS examples described at this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2] are all provided with a C, C++, or SystemC test bench. These examples can be copied and modified to create a C test bench. They include C functions using arbitrary precision data types.

C Language Support for Synthesis

Understanding what is supported for synthesis is important part of the Vivado HLS UltraFast design methodology. Vivado HLS provides comprehensive support for C, C++, and SystemC. Everything is supported for C simulation; however, it is not possible to synthesize every description into an equivalent RTL implementation.

The two key principles to keep in mind when reviewing the code for implementation in an FPGA are:

- An FPGA is a fixed size resource. The functionality must be fixed at compile time. Objects in hardware cannot be dynamically created and destroyed.
- All communication with the FPGA must be performed through the input and output ports. There is no underlying Operating System (OS) or OS resources in an FPGA.

Unsupported Constructs

System Calls

System calls are not supported for synthesis. These calls are used to interact with the OS upon which the C program executes. In an FPGA there is no underlying OS to communicate with. Example of systemcalls are `time()` and `printf()`.

Some commonly used functions are automatically ignored by Vivado HLS and there is no requirement to remove them from the code. These functions are:

- `abort()`
- `atexit()`
- `exit()`
- `fprintf()`
- `printf()`
- `perror()`
- `putchar()`
- `puts()`

An alternative to removing any unsupported code is to guard it from synthesis. The `__SYNTHESIS__` macro is automatically defined by Vivado HLS when synthesis is performed.

This macro can be used to include code when C simulation is run, but exclude the code when synthesis is performed.

```
#ifndef __SYNTHESIS__
  // The following code is ignored for synthesis
  FILE *fp1;
  char filename[255];
  sprintf(filename,Out_apb_%03d.dat,apb);
  fp1=fopen(filename,w);
  fprintf(fp1, %d \n, apb);
  fclose(fp1);
#endif
```

Note: Only use the `__SYNTHESIS__` macro in the code to be synthesized. Do *not* use this macro in the test bench, because it is not obeyed by C simulation or C RTL co-simulation.

If information is required from the OS, the data must be passed into the top-level function for synthesis as an argument. It is then the task of the remaining system to provide this information to the synthesized IP block. This can typically be done by implementing the data port as an AXI4-Lite interface connected to a CPU.

Dynamic Objects

Dynamic objects cannot be synthesized. The function calls `malloc()`, `alloc()`, pre-processor `free()`, and C++ `new` and `delete` dynamically create or destroy memory resources that exist in the OS memory map. The only memory resources available in an FPGA are block RAMs and registers. Block RAMs are created when arrays are synthesized and the values in the arrays must be maintained over one or more clock cycles. Registers are created when the value stored by a variable must be maintained over one or more clock cycles. Arrays of a fixed size or variables must be used in place of any dynamic memory allocation.

As with restrictions on dynamic memory usage, Vivado HLS does not support (for synthesis) C++ objects that are dynamically created or destroyed. This includes dynamic polymorphism and dynamic virtual function calls. New functions, which would result in new hardware, cannot be dynamically created at run time.

For similar reasons, recursion is not supported for synthesis. All objects must be of a known size at compile time. Limited support for recursion is provided when using templates.

Except for standard data types, such as `std::complex`, C++ Standard Template Libraries (STLs) are *not* supported for synthesis. These libraries contain functions which make extensive use of dynamic memory allocation and recursion.

SystemC Constructs

An `SC_MODULE` cannot be nested inside, or derived from, another `SC_MODULE`.

The `SC_THREAD` construct is not supported (however, `SC_CTHREAD` is).

Constructs with Limited Support

Top-Level Function

Templates are supported for synthesis but are not supported for use on the top-level function.

A C++ class object cannot be the top-level for synthesis. The class must be instantiated into a top-level function.

Pointers to pointers are supported for synthesis but not when used as an argument to the top-level function.

Pointer Support

Vivado HLS supports pointer casting between native C types but does not support general pointer casting, such as casting between pointers to differing structure types.

Vivado HLS supports pointer arrays, provided that each pointer points to a scalar or an array of scalars. Arrays of pointers cannot point to additional pointers.

Recursion

Recursion in an FPGA is only supported through the use of templates. The key to performing recursion in synthesis is the use of a termination class, with a size of one, to implement the final call in the recursion.

Memory Functions

The `memcpy()` and `memset()` are both supported with the limitation that `const` values must be used.

- `memcpy()`: used for bus burst operation or array initialization with const values. The memcpy function can only be used to copy values to or from arguments to the top-level function.
- `memset()`: used for aggregate initialization with constant set value.

Any code which is not supported for synthesis, or for which only limited support is provided, must be modified before it can be synthesized.

More complete details on language support and are provided at this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2].

Using Hardware Optimized C Libraries

Vivado HLS provides a number of C libraries for commonly used C functions. The functions provided in the C libraries are generally pre-optimized to ensure high-performance and result in an efficient implementation for when synthesized.

"High-Level Synthesis C Libraries," available at this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2], provides extensive details on all the C libraries provided with Vivado HLS, however, it is highly recommended to have an appreciation of what C functions are available in the C libraries as part of your methodology.

Vivado HLS includes the following C libraries:

- Arbitrary Precision Data Types
- HLS Stream Library
- Math Functions
- Linear Algebra Functions
- Digital signal processing (DSP) Functions
- Video Functions
- IP Library

Understanding Vivado HLS

In it crucial to understand some key HLS concepts before reviewing the subsequent sections in this guide which address optimizing C-based IP. This section provides a brief overview of those concepts.

Measuring Performance

Vivado HLS quickly creates the most optimum implementation based on its own default synthesis behavior and constraints. The clock period is the primary constraint and Vivado HLS uses this along with the target device specifications to determine how many operations can be performed within a clock cycle.

After satisfying the clock frequency constraint, the performance metrics used by Vivado HLS, and in order of optimization importance, are:

- **Initiation Interval (II)**: This is the number of clock cycles between new inputs. This represents the throughput and how quickly the design reads the next input and processes it.

- **Latency**: This is the number of clock cycles required to generate the output. After the minimum interval has been achieved, or if no internal target has been specified, Vivado HLS seeks to minimize the latency.
- **Area**: After the minimum latency has been achieved, Vivado HLS seeks to minimize the area.

The performance metrics are reported for the entire function. For example, if the function has a scalar input, II=3 means the function is processing 1 sample every 3 clock cycles. However, if the function has an input array of N elements, II=N means N elements are processed every N clocks: a rate of one sample per clock.

Optimization directives can be used to direct Vivado HLS to create a design which prioritizes the above metrics: for example, force a reduction in area or latency over the throughput. In the absence of any optimization directives, Vivado HLS uses these goals and the default synthesis behavior outlined below to create the initial design.

Interface Synthesis

The arguments to the top-level function are synthesized into data ports with an optional IO protocol. An IO protocol is one or more signals associated with the data port to automatically synchronize data communication between the data port and other hardware blocks in the system.

For example, in an handshake protocol, the data port is accompanied by a valid port to indicate when the data is valid for reading or writing and an acknowledge port to indicate the data has successfully been read or written.

A full description of the IO protocols is provided at this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2], but these interfaces include AXI, AXI4-Stream and AXI4-Lite interfaces, allowing the IP to be easily integrated into the system using IP integrator.

In addition, an IO protocol is implemented by default for the top-level function itself. This protocol controls when the IP can start operation and indicates when the IP has completed its operation or is ready for new input data. This optional IO protocol can be implemented as an AXI4-Lite interface, allowing the design to be controlled from a microprocessor.

Function Synthesis

Functions are synthesized into hierarchical blocks in the final RTL design. Each function in the C code will be represented in the final RTL by a unique block. In general, optimizations stop at function boundaries; some optimization directives have a recursive option or behavior which allows the directive to take effect across function boundaries.

Functions can be inlined using an optimization directive. This permanently removes the function hierarchy and can lead to better optimization of the logic. Functions can also be pipelined to improve their throughput performance.

Functions are scheduled to execute as early as possible. The following examples shows two functions, `foo_1` and `foo_2`.

```
void foo_1 (a,b,c,d,*x,*y) {
  ...
  func_A(a,b,&x);
  func_B(c,d,&y);
}
```

In function `foo_1`, there is no data dependency between functions `func_A` and `func_B`. Even though they appear serially in the C code, Vivado HLS implements an architecture where both functions start to process data at the same time in the first clock cycle.

```
  void foo_2 (a,b,c,*x,*y) {
    int *inter1;
     ...
     func_A(a,b,&inter1,&x);
     func_B(c,d,&inter1,&y)
}
```

In function `foo_2`, there is a data dependency between the functions. Internal variable `inter1` is passed from `func_A` to `func_B`. In this case, Vivado HLS must schedule function `func_B` to start only after function `func_A` is finished.

Loop Synthesis

Loops by default are left "rolled." This means that Vivado HLS synthesizes the logic in the loop body once and then executes this logic serially until the loop termination limit is reached. Loops can be "unrolled" allowing all the operations to occur in parallel but creating multiple copies of the loop hardware, or they can be pipelined to improve performance.

Loops are always scheduled to execute in order. In the following example, there is no dependency between loop `SUM_X` and `SUM_Y`, however, they will always be scheduled in the order they appear in the code.

```
#include "loop_sequential.h"

void loop_sequential(din_t A[N], din_t B[N], dout_t X[N], dout_t Y[N],
              dsel_t xlimit, dsel_t ylimit) {

     dout_t X_accum=0;
     dout_t Y_accum=0;
     int i,j;

     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;
     }
}
```

Example 4-1: **Sequential Loops**

Logic Synthesis

By default, the logic within functions and loops is always synthesized to execute as early as possible. Vivado HLS always seeks to minimize the latency and while achieving design constraints. Operators in the C code, such as +, *, and /, are synthesized into hardware cores. Vivado HLS automatically selects the most appropriate core to achieve the synthesis goals. The optimization directive RESOURCE can be used to explicitly specify which hardware core is used to implement the operation.

Array Synthesis

By default, Vivado HLS synthesizes arrays into block RAM.

In an FPGA, block RAM is provided in blocks of 18K-bit primitive elements. Each block RAM uses as many 18K primitive elements as required to implement the array. For example, an array of 1024 `int` types requires 1024 * 32-bit = 32768 bits of block RAM, which requires 32768/18000 = 1.8 18K block RAM primitives to implement the block RAM. Although Vivado HLS reports that each array is synthesized into one block RAM, the block RAM might contain multiple 18K primitive block RAM elements.

By default, Vivado HLS makes no attempt to group smaller block RAMs into a single large block RAM or partition large block RAMs into smaller block RAMs. However, this is possible using optimization directives. Vivado HLS might automatically partition small arrays into individual registers to improve the quality of results.

Vivado HLS automatically determines whether to use a single or dual-port block RAM based on the synthesis goals. For example, Vivado HLS uses a dual-port block RAM if this helps to minimize the interval or latency. To explicitly specify whether to use a single or dual-port block RAM, you can use the RESOURCE optimization directive.

Pipelining Functions, Loops, and Tasks

The key to achieving a high performance design is to use the PIPELINE and DATAFLOW optimization directives to pipeline functions, loops and tasks.

The following figure shows a conceptual explanation of pipelining. Without pipelining, the operations execute sequentially until the function completes. Then the next execution of the function, or the next transaction, is performed. With pipelining, the next transaction starts as soon as the hardware resource becomes available.

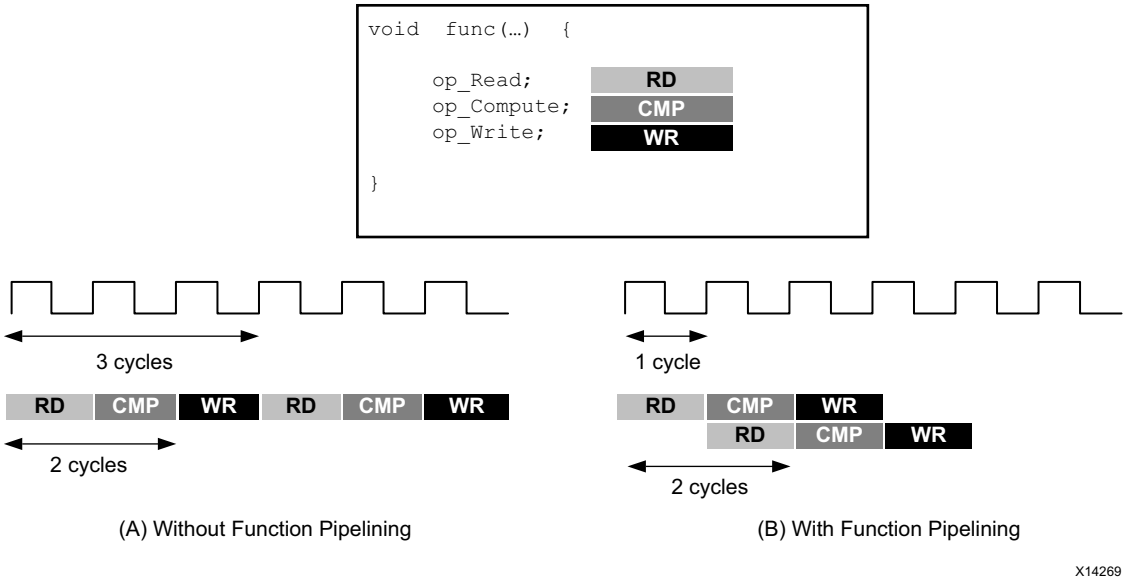

Figure 4-3: **Pipelining Behavior**

The PIPELINE directive can be used on functions or loops to improve the throughput (minimize the II) with minimal area overhead.

Functions and loops are considered tasks. The DATAFLOW directive can be used to "pipeline" tasks, allowing them to execute concurrently if data dependencies allow.

Figure 4-4 shows a conceptual view of task pipelining. After synthesis, the default behavior is to execute and complete `func_A`, then `func_B`, and finally `func_C`. However, you can use the DATAFLOW optimization directive to schedule each function to execute as soon as data is available.

In this example, the original function has a latency and interval of 8 clock cycles. When you use dataflow optimization, the interval is reduced to only 3 clock cycles. The tasks shown in this example are functions, but you can perform dataflow optimization between functions, between functions and loops, and between loops.

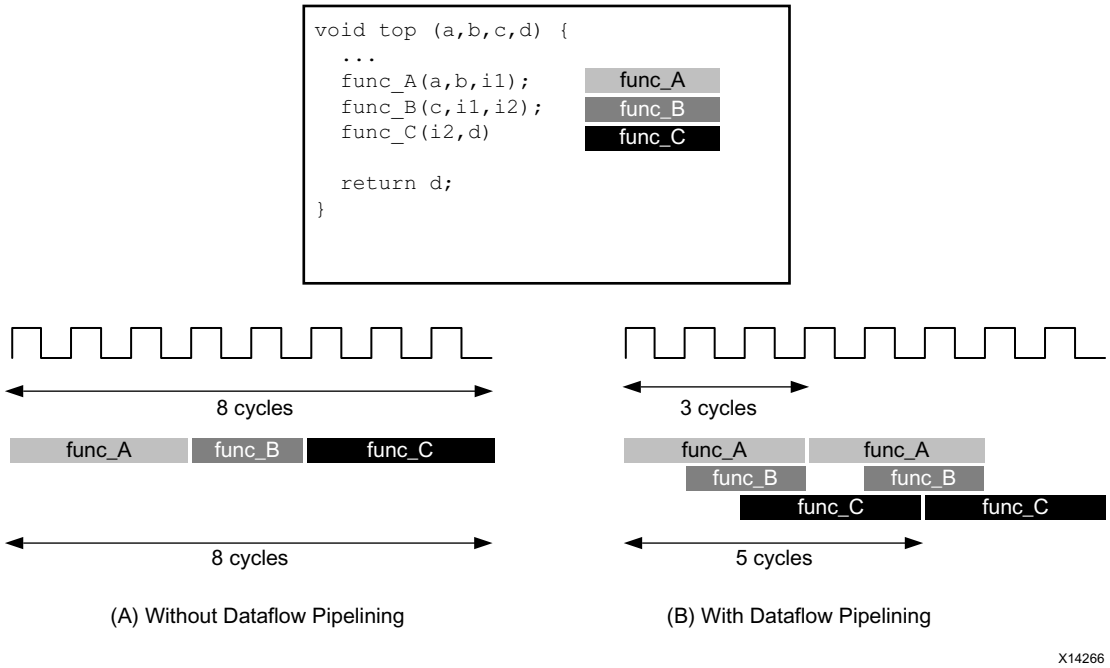

Figure 4-4: **Dataflow Optimization**

Vivado HLS Resources

The Vivado HLS Design Hub in the Documentation Navigator provides easy access to learn more about Vivado HLS:

- QuickTake videos explaining the operation
- Tutorials on every aspect of the design flow
- Vivado HLS User Guide
- Multiple Application Notes

For more information about design hubs, refer to Using the Documentation Navigator.

Optimization Methodology

In addition to the default synthesis behavior discussed in the previous section, Vivado HLS provides a number of optimization directives and configurations that are used to direct synthesis towards a desired outcome. This section outlines a general methodology for optimizing your design for high-performance.

There are many possible goals when trying to optimize a design using Vivado HLS. The methodology assumes you want to create a design with the highest possible performance, processing one sample of new input data every clock cycle, and so addresses those optimizations before those used for reducing latency or resource.

The next section, The HLS Optimization Methodology, addresses how to apply the methodology described here for different C code architectures.

Detailed explanations of the optimizations discussed here are provided in the following sections of the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2]:

- "Managing Interfaces," available at this link.
- "Design Optimization," available at this link.

It is highly recommended to review the methodology and obtain a global perspective of high-level synthesis optimization before reviewing the details of specific optimization.

The HLS Optimization Methodology

The optimization methodology for Vivado HLS is shown in Figure 4-5. It cannot be overstated how important it is to first verify your C code is functionally correct. The remaining steps are explained below and can be summarized as: determine your interfaces, pipeline the design, address issues which prevent optimal pipelining by optimizing data structures and then address any latency and area concerns.

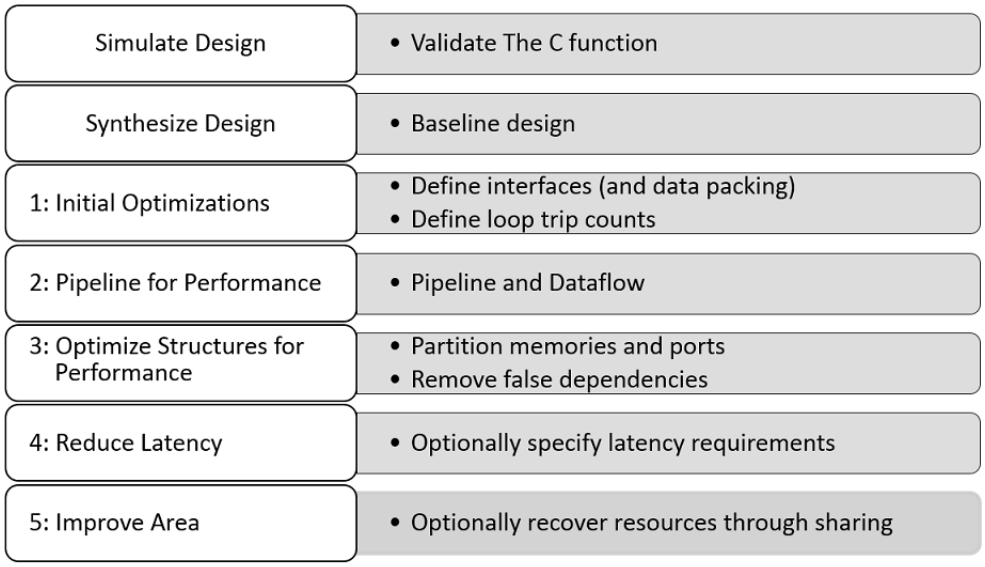

Figure 4-5: **HLS Optimization Methodology**

The following is a complete list of optimization directives. This list shows the Tcl commands on the left side and the equivalent pragma directives, which can be placed directly in the C code, on the right:

```
set_directive_allocation           - Directive ALLOCATION
set_directive_array_map            - Directive ARRAY_MAP
set_directive_array_partition      - Directive ARRAY_PARTITION
set_directive_array_reshape        - Directive ARRAY_RESHAPE
set_directive_data_pack            - Directive DATA_PACK
set_directive_dataflow             - Directive DATAFLOW
set_directive_dependence           - Directive DEPENDENCE
set_directive_expression_balance   - Directive EXPRESSION_BALANCE
set_directive_function_instantiate - Directive FUNCTION_INSTANTIATE
set_directive_inline               - Directive INLINE
set_directive_interface            - Directive INTERFACE
set_directive_latency              - Directive LATENCY
set_directive_loop_flatten         - Directive LOOP_FLATTEN
set_directive_loop_merge           - Directive LOOP_MERGE
set_directive_loop_tripcount       - Directive LOOP_TRIPCOUNT
set_directive_occurrence           - Directive OCCURRENCE
set_directive_pipeline             - Directive PIPELINE
set_directive_protocol             - Directive PROTOCOL
set_directive_reset                - Directive RESET
set_directive_resource             - Directive RESOURCE
set_directive_stream               - Directive STREAM
set_directive_top                  - Directive TOP
set_directive_unroll               - Directive UNROLL
```

Configurations modify the default synthesis behavior. There are no pragma equivalents for the configurations. In the GUI, configurations are set using the menu **Solution > Solution Settings > General**. A complete list of the available configurations is:

```
config_array_partition          - Config the array partition
config_bind                     - Config the options for binding
config_compile                  - Config the optimization
config_dataflow                 - Config the dataflow pipeline
config_interface                - Config command for io mode
config_rtl                      - Config the options for RTL generation
config_schedule                 - Config scheduler options
```

Having a list of all the optimization directives and synthesis configurations is good. Having a methodology to use them is better.

Step 1: Initial Optimizations

The following table shows the first directives you should think about adding to your design.

Table 4-1: **Optimization Strategy Step 1: Initial Optimizations**

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 which 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 IO 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 IO protocol helps determine what can be achieved by synthesis it is recommended to use the INTERFACE directive 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 case 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 Vivado HLS 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.

Note: Loops with variable bounds cannot be unrolled completely and prevents function and loops above them in the hierarchy from being pipelined. This is discussed in the next section.

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

Step 2: Pipeline for Performance

The next stage in creating a high performance design is to pipeline the functions, loops, and tasks. The following table shows the directives you can use for pipelining.

Table 4-2: **Optimization Strategy 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 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.

The basic strategy at this point in the optimization process is to pipeline the tasks (functions and loops) and as much as possible. For detailed information on which functions and loops to pipeline and where to apply the DATAFLOW directive, refer to Optimization Strategies.

For designs with many loops or many nested loops, the compile configuration provides a way to automatically pipeline all the loops in the design based on their loop iteration count. For more information, refer to this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2].

Although not commonly used, you can also apply pipelining at the operator level. For example, wire routing in the FPGA can introduce large and unanticipated delays that make it difficult for the design to be implemented at the required clock frequency. In this case, you can use the RESOURCE directive to pipeline specific operations, such as multipliers, adders and block-RAM.

The RESOURCE directive specifies which hardware cores to use to implement the operations in the C code. Specifying that a resource is implemented with a latency greater than 1 causes Vivado HLS to use additional pipeline stages for the operation. RTL synthesis can use these additional pipeline stages to improve the overall timing.

The following operations support pipelined implementations:

- Standard arithmetic operations for which there is a multi-stage (*nS) core available
- Floating-point operations
- Arrays implemented with a block RAM

Step 3: Optimize Structures for Performance

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 using other optimization directives.

The following example shows a case where an optimization directive is used to improve the performance of pipelining. In this initial example, the PIPELINE directive is added to a loop to improve the performance of the loop.

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

When the code above is synthesized the following message is output:

```
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.
I
```

When pipelining fails to meet the required performance, the key to addressing the issue is to examine the design in the analysis perspective. Details on using the Analysis Perspective are provided at this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2]. The view of this design example in the analysis perspective is shown in the figure below.

- The memory (block-RAM) accesses are highlighted in the figure. These correspond to the array `mem` in the code above.
- Each access takes two cycles: one to generate the address and one to read the data.
- Only two memory reads can start in cycle C1 because a block RAM only has a maximum of two data ports.
- The third and forth memory reads can only start in cycle C2.
- The earliest the next set of memory reads can occur is starting in cycle C3. This means the loop can only have an II=2: the next set of inputs to the loop can only be read every two cycles.

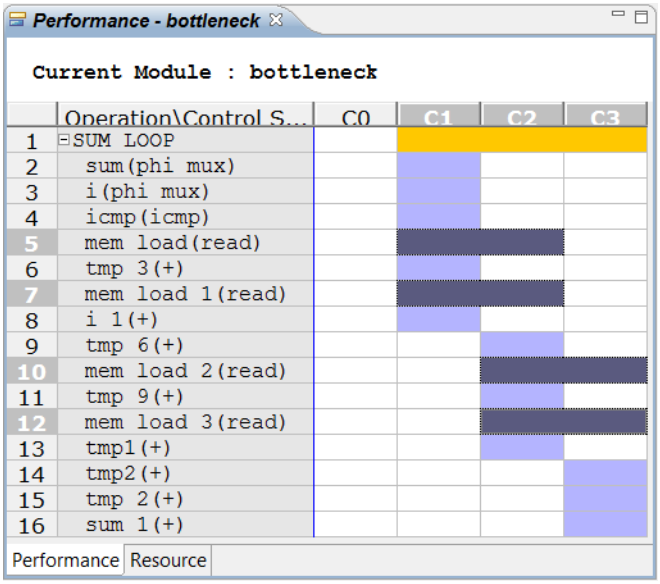

Figure 4-6: **Pipelining Fails Due to Too Few Ports**

The memory port limitation issue can be solved by using the ARRAY_PARTITION directive on the mem array. This directive partitions arrays into smaller arrays providing more data ports and improves the data structures to allow a higher performance pipeline.

With the additional directive shown below, array `mem` is partitioned into two dual port memories so that all four reads can occur in one clock cycle. There are multiple options to partition an array. In this case cyclic partitioning with a factor of 2 ensures the first partition contains elements 0,2,4 etc. from the original array and the second partition contains elements 1,3,5, etc. With a dual-port block-RAM this allows elements 0,1,2 and 3 to be read in a single clock cycle.

```
#include "bottleneck.h"

dout_t bottleneck(din_t mem[N]) {
#pragma HLS ARRAY_PARTITION variable=mem cyclic factor=2 dim=1

  dout_t sum=0;
  int i;

  SUM_LOOP: for(i=3;i<N;i=i+4)
#pragma HLS PIPELINE
  sum += mem[i] + mem[i-1] + mem[i-2] + mem[i-3];

  return sum;
}
```

Other such issues might be encountered when trying to pipeline loops and functions. The following table lists the directives which are likely to address these issues by helping to reduce bottlenecks in data structures.

Table 4-3: **Optimization Strategy 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 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, the verbosity of the output messages, and specify if II should be relaxed in pipelined tasks in order to achieve timing.
CONFIG_UNROLL	Allows all loops below the specified number of loop iterations to be automatically unrolled.

In addition to the ARRAY_PARTITION directive, the configuration for array partitioning can be used to automatically partition arrays.

The configuration for compile is used to automatically pipeline loop hierarchies. The DEPENDENCE directive might be required to remove implied dependencies when pipelining loops. Such dependencies will be reported by message SCHED-68.

```
@W [SCHED-68] Target II not met due to carried dependence(s)
```

The INLINE directive removes function boundaries. This can be used to bring logic or loops up one level of hierarchy. It might be more efficient to pipeline the logic in a function by including the logic in the function above it and it can be easier to dataflow a series of loops with other loops by raising them up the hierarchy.

The UNROLL directive might be required for cases where a loop cannot be pipelined with the required initiation interval. If a loop can only be pipelined with II=4 it will constrain the other loops and functions in the system to be limited to II=4. In some cases, it might be worth unrolling the loop - creating more logic, but removing a potential bottleneck.

The schedule configuration is used to increase the verbosity of the scheduling messages and to control the effort levels in scheduling. When the verbose option is used Vivado HLS lists the critical path when scheduling is unable to meet the constraints.

In general, there are few cases where increasing the schedule effort improves the scheduling but the option is provided. If optimization directives and configurations cannot be used to improve the initiation interval it might require changes to the code. Examples of this are discussed at this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2].

Step 4: Reduce Latency

When Vivado HLS finishes minimizing the initiation interval it automatically seeks to minimize the latency. The optimization directives listed in the following table 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.

Table 4-4: **Optimization Strategy Step 4**

Directive	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.

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 less number of clock cycles a design will take to complete.

Step 5: Reducing Area

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 Vivado HLS cannot determine whether the tasks in the design are streaming data, Vivado HLS 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.

The following table lists the other directives to consider when attempting to minimize the resources used to implement the design.

Table 4-5: **Optimization Strategy Step 5**

Directive	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.
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.

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 1 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.

IMPORTANT: *Optimization directives are only applied within the scope in which they are specified.*

If the ARRAY_PARITION 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 re-combines 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 OCCURENCE directive is used to indicate this logic can be optimized to execute at a lower rate.

Optimization Strategies

The Optimization Methodology is generally applicable to all types of C code. The key optimization directives for obtaining a high-performance design are the PIPELINE and DATAFLOW directives. This section discusses in detail how to apply these directives for various architectures of C code.

Fundamentally, there are two types of C functions. Those which are frame based and those which 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.

Frame Based C Code

An example outline of Frame based C code is shown below. The primary characteristic of this coding style is that the function processes multiple data samples - a frame of data - during each transaction (a transaction is considered one complete execution of the C function).

```
void foo(
  data_t in1[HEIGHT][WIDTH],
  data_t in2[HEIGHT][WIDTH],
  data_t out[HEIGHT][WIDTH] {


Loop1: for(int i = 0; i < HEIGHT; i++) {
  Loop2: for(int j = 0; j < WIDTH; j++) {
    out[i][j] = in1[i][j] * in2[i][j];
    Loop3: for(int k = 0; k < NUM_BITS; k++) {
  }
}
}
```

The data can typically be provided as arrays but it could be provided as a pointer or an `hls::stream`. Pointers can be accessed multiple time using pointer arithmetic and an `hls::stream` can be accessed multiple times within a function.

Another characteristic of frame based coding style is that the data will typically be accessed and processed using loops. The code above shows a classic example of this.

When seeking to pipeline any C code, you want to place the pipeline directive at the level where a sample of data is processed. Discussing this for each of the levels in the above example helps explain the best to place the pipeline directive.

Function Level: The function accepts a frames of data as input (in1 and in2). If the function is pipelined with II=1 - read a new set of inputs every clock cycle - this informs the tool to read all HEIGHT*WIDTH values of in1 and in2 in a single clock cycle.

It is unlikely this is the design you want.

When a PIPELINE directive is applied, all loops in the hierarchy below this level, everything below foo in this case, must be unrolled. This is a requirement for pipelining: there cannot be sequential logic inside the pipeline. This would create HEIGHT*WIDTH*NUM_ELEMENT copies of the logic: a large design.

The arrays can be implemented as multiple types of interface:

- Block-RAM interface (default)
- AXI interface
- AXI4-Lite interface
- AXI4-Stream interface
- FIFO interface

Since the data is accessed in a sequential manner, as an AXI4-Stream interface. two-way handshake or FIFO interface. A block-RAM interface can be implemented as a dual-port interface supplying two samples per clock. The other interface type can only supply one sample per clock. This would create a bottleneck.

Step 3 of the HLS Optimization Methodology allows you to get past this bottleneck. To access all data values in the same clock cycle the array must be partitioned into its individual elements to create HEIGHT*WIDTH ports (or half that number if each port is a dual-port block-RAM interface) allowing all ports be read in the same clock cycle. Similarly for the output port.

Note: For more information about the optimization methodology, refer to Optimization Methodology

It would be a highly parallel design, but it would be large.

Loop1 Level: The logic in Loop1 processes an entire row of the two-dimensional matrix. Placing the PIPELINE directive here would create a design which seeks to process one row in each clock cycle. Again, this would unroll the loops below and create additional logic.

A large parallel design is not as fast and large as the first option.

Loop2 Level: The logic in Loops seek to process one sample from the arrays. This is the level to pipeline if the design is to process one sample per clock cycle.

This causes Loop3 to be completely unrolled but since Loop2 is processing one sample per clock, this is a requirement and typically necessary. In a typical design, the logic in Loop3 is typically a shift register or processing bits within a word. To execute at one sample per clock, you want these processes to occur in parallel and hence you want to unroll the loop.

This design processes one data sample per clock and creates parallel logic only where required to achieve this level of data throughput.

Loop3 Level: As stated above, in this case the logic in Loop3 will typically be doing bit-level or data shifting tasks - the level above that is operating on each data sample. For example, if Loop3 contained a shift register operation and Loop3 is pipelined, this would inform the tool to shift one data value every clock cycle. The design would only return to the logic in Loop2 and read the next inputs when all samples have been shift.

The ideal location to pipeline in this example is Loop2.

When dealing with frame based code you will want to pipeline at the loop level and typically pipeline the loop which operates at the level of a sample. If in doubt, place a print command into the C code and use C simulation to confirm this is the level you wish to execute on each clock cycle.

As detailed above, it is typical in a Frame based design to use the ARRAY_PARTITION directive to partition arrays into smaller blocks (or multiple ports for arrays on the interface) to remove bottlenecks to performance.

Sample Based C Code

An example outline of Sample based C code is shown below. The primary characteristic of this coding style is the function processes a single data sample during each transaction.

```
void foo (data_t *in, data_t *out) {

  static data_t acc;

  Loop1: for (int i=N-1;i>=0;i--) {
  acc+= ..some calculation..;
  }

  *out=acc>>N;
}
```

In a sample-based function the data is provided as a scalar, a pointer or an `hls::stream` variable.

A pointer or `hls::stream` may be accessed multiple times within a function but in a sample-based function they should only be accessed once.

Another characteristic of sample based coding style is that the function often contains a static variable: a variable whose value must be remembered between invocations of the function, such as an accumulator or sample counter.

To achieve an II=1, reading one data value each clock cycle, the function must be pipelined. This will unroll any loops and create additional logic but there is no way around this. If `Loop1` is pipelined, it will take a minimum of N clock cycles to complete. Only then can the function can read the next `x` input value.

When dealing with C code which processes at the sample level, the strategy is always to pipeline the function. Since loops in a sample based design are typically operating on arrays performing a shift register function, it is not uncommon to partition these arrays into individual elements to ensure all samples are shifted in a single clock cycle: else the shift operation will be limited to reading and writing the samples to a dual-port block-RAM.

The solution here is to pipeline function `foo`. Doing so results in a design which processes one sample per clock.

RTL Verification

The RTL verification process within Vivado HLS is fully automated. During RTL/C co-simulation the same C test bench used in C simulation is re-used and the synthesized function is replaced by the RTL design. The sequencing of data, into and out of the RTL design using the correct interface protocols, is performed automatically by Vivado HLS.

Because the C test bench is re-used, an RTL test bench does not need to be created.

There are a few design choices which might prevent RTL/C co-simulation being available. To perform a RTL/C co-simulation the following conditions must be true.

- The top-level function must be synthesized using an `ap_ctrl_hs` or `ap_ctrl_chain` block-level interface.
- Or the design must be purely combinational.
- Or the top-level function must have an initiation interval of 1.
- Or the 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 co-simulation 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 ***
```

IP Packaging

Once the design is completed, an IP Package suitable for the IP Catalog is created using the Export RTL feature of Vivado HLS. For design which include an AXI4-Lite interface, the IP package contains the necessary software driver files to program the interface.

Vivado HLS provides multiple packing options. To use a productive IP Integrator methodology, the IP Catalog format should be used and the interfaces should be AXI interfaces.

Design Analysis and Optimization

A necessary part of any design methodology is using a productive process for design analysis and improvement. The process for using Vivado HLS for C simulation, C debug, synthesis, analysis, RTL verification and IP packaging is described at this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2].

The process for generating the design and improving its performance can be summarized as:

- Simulate the C code and validate the design is correct.
- Synthesize an initial design.
- Analyze the design performance.
- Create a new solution and add optimization directives.
- Analyze the performance of the new solution
- Continue creating new solutions and optimization directives until the requirements are satisfied.
- Verify the RTL is correct.
- Package the design as IP and include it into your system.

The most productive methodology is one that uses C simulation to both validate the design and confirm the results are correct before synthesis. The benefit of C simulation speed is the major advantage of a high-level design flow. Confirming the correctness of the C design is a much more productive use of your time than debugging performance issues which turn out to be due an incorrect specification.

Ensuring Useful Reports

After the initial synthesis results are achieved, the first step is to review the results. If the synthesis report contains any unknown values (shown as a question mark "?") they must be addressed. To determine if optimization directives improve the design performance, it is crucial to be able to compare the solutions: the latency must have known values for comparison.

If a loop has variable bounds, Vivado HLS cannot determine the number of iterations for the loop to complete. Even if the latency of one iteration of the loop is known, it cannot determine the latency to complete all iterations of the loop due to the variable bounds.

Review the loops in the design. In the synthesis report, review the loops in the **Latency > Details > Loops** section. Start with the lowest level loop in the loop hierarchy which reports an unknown latency as this unknown propagates up the hierarchy. The loop or loops can be in lower levels of hierarchy. Review the **Latency > Details > Instance** section of the report to determine if any sub-functions show unknown values. Open the report for any functions which show unknown latency values and repeat the process until the loop or loops have been identified.

An alternative to using the synthesis report is to use the Analysis perspective.

After you identify the variable bound loops, add a LOOP_TRIPCOUNT directive to specify the loop iteration count, or use assertions in the C code to specify the limits. For more information, refer to this link in the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2].

If you are using the LOOP_TRIPCOUNT directive, consider adding the directive to the source code as a `pragma`, because this directive is required in every solution.

If you are aware of other loops with variables bounds, provide iteration limits for these loops, otherwise repeat synthesis and use the same bottom up process until the top-level report contains real numbers.

Design Analysis

Design analysis can be performed using three different techniques.

- The synthesis reports.
- The analysis perspective.
- RTL simulation waveforms.

TIP: *Before analyzing the results, review the console window or log file to see what optimizations where performed, skipped or failed.*

The synthesis reports and analysis perspective can be used to analyze the latency, interval and resource estimates. If there is now more than one solution, use the compare reports button in the GUI to compare the solutions side-by-side. This feature, like the analysis perspective, is only available in the GUI but remember that projects created using batch-mode can be opened for analysis in the GUI using `vivado_hls -p project_name`.

Again, a hierarchical approach is useful. Start at the top-level and determine which tasks contribute the most to either the latency, interval, or area, and examine those tasks in more detail. Recursively move down the hierarchy until you find a loop or function which you think could or should be performing better to achieve your goals. When these functions or loops are improved, the improvements will ripple up the hierarchy.

The analysis perspective allows for much easier migration up and down the design hierarchy than the synthesis reports. In addition the analysis perspective provides a detailed view of the scheduled operations and resource usage which can be cross-correlated with the C code.

It can be useful when using the detailed schedule views in the analysis perspective to first look at the macro-level behavior before diving into the details. The operations are typically listed in the order the code executes. Keep in mind, Vivado HLS will try to schedule everything into clock cycle 1 and be finished in 1 clock cycle if it can.

- If you see a general drift in the operations from the top-left to the bottom-right, it is probably due to data dependencies or the execution of tasks inherent in the code. Each needs the operation before to complete before it can start.
- If you see operations scheduled one after then other, then the sudden execution of many items in parallel, or vice-versa, it probably indicates a bottleneck (such as I/O ports or RAM ports) where the design has to wait, and wait, then everything can execute in parallel.

In addition to the synthesis reports and analysis perspective, the RTL simulation waveforms can be used to help analyze the design. During RTL verification the trace files can be saved and viewed using an appropriate viewer. Refer to the tutorial on RTL verification in the *Vivado Design Suite Tutorial: High-Level Synthesis* (UG871) [Ref 3] for more details. Alternatively, export the IP package and open the Vivado RTL project in the `project_name/solution_name/impl/ip/verilog or vhdl` folder. If C/RTL co-simulation has been executed, this project will contain an RTL test bench.

Be very careful investing time in using the RTL for design analysis. If you change the C code or add an optimization directive you will likely get a very different RTL design with very different names when synthesis is re-executed. Any time spent in understanding the RTL details will likely need repeated every time a new design is generated and very different names and structures are used.

In summary, work down the hierarchy to identify tasks which could be further optimized.

Design Optimizations

Before performing any optimizations it is recommended to create a new solution within the project. Solutions allow one set of results to be compared against a different set of results. They allow you to compare not only the results, but also the log files and even output RTL files.

The basic optimization strategy for a high-performance design is:

- Create an initial or baseline design.
- Pipeline the loops and functions.
- Address any issues which limit pipelining, such as array bottlenecks and loop dependencies (with ARRAY_PARTITION and DEPENDENCE directives).
- Apply the DATAFLOW optimization to execute loops and functions concurrently.
- It might sometimes be necessary to make adjustments to the code to meet performance.
- Reduce the size of the dataflow memory channels and use the ALLOCATION and RESOUCES directives to further reduce area.

In summary, the goal is to always meet performance first, before reducing area. If the strategy is to create a design with fewer resources, then simply omit the steps to improving performance.

Throughout the optimization process it is highly recommended to review the console output (or log file) after synthesis. When Vivado HLS cannot reach the specified performance goals of an optimization, it will automatically relax the goals (except the clock frequency) and create a design with the goals which can be satisfied. It is important to review the output from the synthesis to understand what optimizations have been performed.

For specific details on applying optimizations, refer to the *Vivado Design Suite User Guide: High-Level Synthesis* (UG902) [Ref 2].

Chapter 5

System Integration

Overview

As with the development of the shell design, the key to a productive system integration methodology is to make use of the Vivado® IP Catalog and IP integrator. If the earlier steps in the high-level design methodology were followed, this stage of the design process includes the following:

- A pre-verified shell. The board-level interface has been developed and successfully verified.
- A collection of pre-verified IP blocks representing the core functionality of the design, packaged for the Vivado IP Catalog.
- AXI Interfaces used for the IP level interfaces, enabling the Designer Assistance feature of IP integrator to automate design creation.
- A system-level test bench, already created to verify the shell.

The system components, developed and verified in parallel, are now ready for system integration.

After the initial system integration, you have everything required to automate this entire flow and easily generate additional new designs.

Initial System Integration

The design integration process can be summarized as follows:

1. Create a new Vivado project based on the shell design
2. Add all the IP blocks, using IP integrator to connect the IP
3. Verify the system and processing the design through implementation to an FPGA bitstream.

System Integration Project

A new system design integration project is created using one of the reference shell designs as follows:

1. Open the Vivado project for the shell design and select **File > Save Project As** to save the shell design in a new project.
2. Create a new Vivado RTL project (with no RTL sources) and the same target device or board. Then select **Create Block Design**, and in the console, source the Tcl script saved using `write_bd_tcl` to regenerate the shell Block Design in the new project.
3. Add all the core design IP blocks to the project IP repository.

Automated System Integration

IP integrator provides Designer Assistance to help integrate the system. The recommended strategy is to open the shell design and add all the IP blocks to the canvas. If AXI Interfaces are being used for all IP and shell interfaces, the Designer Assistance feature activates with recommendations for connections.

Designer Assistance recognizes which connections are legal. In addition to simply automating connections, it is able to automatically add any required AXI interconnect logic, such as connecting an AXI4-Stream interface to an AXI memory mapped port. For a complete description of the Designer Assistance feature, refer to this link in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 8].

TRAINING: *If multiple clock domains are used in the design, refer to the* Vivado Design Suite QuickTake Video: Using Multiple Clock Domains in Vivado IP Integrator.

Complete the block design by making the connections that are not supported for designer assistance, such as scalar signals and any non-AXI bus interfaces.

Finally, use the validate design feature to ensure that there are no design rule violations in the design. For designs that use memory mapped interfaces or processors, refer to this link in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 8].

Xilinx® recommends that you use the `write_bd_tcl` command when the design is complete and has been successfully validated. The `write_bd_tcl` command saves everything required to re-generate the completed system in a Tcl file.

System Verification and Implementation

When the system block design is complete in IP integrator, you can process the complete system for verification and implementation by generating output products and creating a top-level HDL wrapper for the design. The complete flow for using IP integrator in this manner is shown in the following sections of the *Vivado Design Suite Tutorial: High-Level Synthesis* (UG871) [Ref 3]:

- Chapter 9, "Using HLS IP in IP Integrator"
- Chapter 10, "Using HLS IP in a Zynq SoC Design"

System level verification is now performed using the same RTL test bench created for the shell verification. The individual parts—the IP blocks and the shell design—have been pre-verified in isolation. The task is now to verify the complete system. Focus your initial efforts in confirming the system-level connections by doing the following:

1. Ensure that the shell is correctly supplying data to the inputs of the first block in the processing pipeline.
2. Ensure that the first block is correctly providing outputs to the next block.

Note: Use minimal amounts of data to ensure that the system level simulation runs as quickly as possible. Your initial focus here should simply be on confirming block connectivity.

After the top-level connectivity is verified a complete and detailed system simulation can be performed.

IMPORTANT: *As discussed in Automated System Integration, below, this advanced methodology provides an effective method to quickly re-design any IP that fails system level verification and then quickly regenerate the entire system in a highly scripted manner.*

When the system is fully verified, the design can be implemented to bitstream. If most of the design IP is created from C/C++ using Vivado HLS, the RTL has been automatically timed to ensure it meets timing during RTL synthesis.

Note: If there are timing paths between IP blocks that fail to meet timing after RTL synthesis, consider using the `INTERFACE` directive during HLS to register the interface ports.

To help improve runtime, consider using the **Out of Context per IP** mode when generating the Output Products. This option generates a synthesized output product, which is cached, and synthesis will only be re-executed if the IP block is changed, reducing the system implementation time.

Automated System Integration

Although it might be a cause for concern that it is only at the system integration stage that a complete system level verification is performed, this is in fact one of the benefits of this methodology. A complete system-level simulation in RTL might take some time to complete and it is typically the execution of many such simulations during the development of a project that is the greatest time sink.

This methodology places importance on:

- Parallel development
- Verification of the design IP using C/C++ simulation to achieve orders of magnitude verification speed increase
- Creation and verification of block-level IP
- Reuse of existing and verified IP

This methodology is so effective because of the high degree of automation provided in the Vivado Design Suite. This section demonstrates how the entire system can be quickly and easily re-created using Tcl scripts, even if an issue is found when the system is integrated.

Vivado Project Automation

All actions performed in the Vivado IDE are captured as Tcl commands in the project journal file. These commands allow you to repeat all actions in batch mode, significantly reducing the time taken to perform the tasks. This auto-generation of Tcl commands allows the following tasks in this methodology to be highly automated:

- Project creation
- Adding IP to the project
- System simulation
- System implementation

The following code sample shows how easy it is to fully automate the process of creating a project, adding IP repositories to the project, creating a block design, and processing the project through to bitstream.

```
# Set project parameters
set my_part xc7z020clg484-1
set my_board_part xilinx.com:zc702:part0:1.0

# Set the paths to auto-adjust to the local directory
# Define project and IP repository locations
set my_files [pwd]
set projdir $my_files/project_1
set repo_dir $my_files/../my_ip/ip
puts "Using project directory $projdir"
puts "Using repository directory $repo_dir"

# Create the Project
set projname project_1
create_project -force $projname $projdir -part $my_part
set_property board_part $my_board_part [current_project]

# Create IP repository
set_property ip_repo_paths $repo_dir [current_fileset]
update_ip_catalog -rebuild

# Create the block design
source ./design_IPI.tcl

# Create output products and HDL wrapper
generate_target all [get_files
$projdir/$projname.srcs/sources_1/bd/$design_name/$design_name.bd]
make_wrapper -files [get_files
$projdir/$projname.srcs/sources_1/bd/$design_name/$design_name.bd] -top
add_files -norecurse
$projdir/$projname.srcs/sources_1/bd/$design_name/hdl/${design_name}_wrapper.v
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1

# Implement the bitstream
launch_runs impl_1 -to_step write_bitstream
wait_on_run impl_1
```

The Tcl commands to perform the above actions are simply copied from the project journal file. Alternately, you can use **Save > Write Project Tcl** to easily script the flow inside Vivado and automate the processes of shell creation, shell verification, and system integration.

Vivado HLS Automation

Vivado HLS creates a Tcl file for every project created using the IDE. The code sample below shows the Tcl commands for all steps in the C design flow: simulating the C code, synthesizing the C code with optimization directives, verifying that the RTL is correct, and creating an IP package and confirming that RTL synthesis meets timing.

```
# Create a project and add files
open_project proj_matrixmul
set_top DESIGN_TOP
add_files matrixmul.cpp
add_files -tb matrixmul_tb.cpp

# Create a solution
open_solution "fast"
set_part {xc7z020clg484-1}
create_clock -period 4 -name default

# Add optimization directives
set_directive_pipeline "cholesky/"
set_directive_array_reshape -type complete -dim 2 "matrixmul" a
set_directive_array_reshape -type complete -dim 1 "matrixmul" b

# Simulate, Synthesize, Verify and package outputs
csim_design
csynth_design
cosim_design
export_design -format ip_catalog -evaluate verilog
```

This automatically generated Tcl file can be edited to provide automation for any part of the C IP development flow. For example, scripts can be created to perform only C simulation. When the design is validated, a complete script like the above can be used to synthesize the design through to a packaged IP.

IP Integrator Automation

The IP integrator `write_bd_tcl` command not only saves a Tcl script to replay your actions, it also optimizes the script to only create the final block design. Simply executing the script will re-create the block design. The block design is re-created using the IP in the IP repository, so if the IP has been updated, the latest IP is used. This level of automation enables you to quickly re-create block designs:

- The shell design can be re-generated in a new design project, modifications performed, and a new shell easily created.
- The shell design can be re-generated in a new verification project, and verification IP easily added to the design.
- The shell design can be re-generated in the system integration project and the core design IP integrated into the system.

Each step in the entire methodology can be re-performed in an efficient productive manner.

Complete System Automation

Further productivity advantages can be realized by executing scripts using a Makefile. A Makefile specifies a set of dependencies. For example, the following tasks must be performed in order:

- Task A: Simulate an IP in C.
- Task B: Synthesize the IP to the IP Catalog.
- Task C: Integrate the IP into the system level.

When a Makefile is used to execute task C, it automatically checks to determine if the outputs from task B are present. If they are not present, it will seek to execute task B and check to see if the outputs from task A are present, and so on.

Using a Makefile to execute Tcl scripts similar to those shown above, it is therefore possible to simply update any IP or the shell design and issue a single command to re-create the entire system, stopping for any of the following:

- To review the results of C simulation
- To re-create the shell design to add verification IP
- To re-synthesize an IP, verify the IP with RTL simulation, and re-build the system.
- After the FPGA has been programmed.

This level of automation links everything in the methodology into a highly productive flow, and is the reason that the parts of the methodology can be performed in parallel and wait until system integration to perform system-level simulations. After the first version of the system is created, the entire generation of the system is fully automated.

Designing for the Future

The final benefit of using a high-level productivity design methodology is how easily derivative designs can be created from the initial design. The two primary enablers of this productivity boost are:

- Developing IP from C
- Using the automated implementation flow

Developing IP from C

In addition to the primary advantages of using C for IP development that are outlined in this guide, another advantage is the easy path to design re-targeting: creating derivative designs from the same source.

In the Vivado HLS scripting example above, the following Tcl commands were used to target a Zynq®-7000 SoC device with a 250 MHz clock:

```
set_part {xc7z020clg484-1}
create_clock -period 4 -name default
```

To create an IP block that is accurately timed for implementation in, for example, a Kintex® UltraScale™ device at 300 MHz, the changes are simply to update the optimization constraints:

```
set_part {xcku025-ffva1156-2-i}
create_clock -period 300MHz -name default
```

Nothing else needs to change. Vivado HLS simply creates a design implemented at the selected frequency in the target technology. The design might take fewer clock cycles to complete in a faster technology (and conversely, more in a slower technology) but no re-coding or re-optimization is necessary.

The greater the content created from C code, the more easily the design can targeted to a newer technology and/or clock frequency. Legacy RTL blocks might still need to be re-implemented to accommodate the different timing parameters.

Automated Implementation Flow

If the FPGA is implemented in a fully scripted flow, you can further improve design reuse and improve productivity when creating derivative design by parametrizing your scripts.

In the scripting examples shown above, the following code is used. This example uses the Vivado script; the same enhancement can be performed on the Vivado HLS and IP integrator scripts.

```
# Set project parameters
set my_part xc7z020clg484-1
set my_board_part xilinx.com:zc702:part0:1.0
```

To set everything in the project by a single top-level project parameters script, change the previous code to match the following:

```
# source project-level parameters
source project_top.tcl
# Set project parameters
set my_part $target_device
set my_board_part $target_board
```

In this example, the contents of `project_top.tcl` are:

```
set target_device xc7z020clg484-1
set target_board xilinx.com:zc702:part0:1.0
```

Modifying this single script causes the project to be re-targeted and re-implemented in a highly automated manner.

Some additional notes about this:

- If you follow the recommendations on C test benches, the C simulation is automatically checked.
- Vivado HLS generates new IP based on the new parameters.
- The RTL created by Vivado HLS is automatically verified by RTL simulation.
- The Vivado project generation scripts create a new project based on the update parameters.
- The IP integrator scripts connect the blocks as before and designer automation will use the most appropriate connections.
- You can choose to pause the flow at the system integration step and modify the design.
- The completed system is implemented to bitstream using the update target device and clock frequency.

Enhancing your Tcl scripts to accommodate a degree of parametrization will greatly enhance the ability of your organization to create design derivatives in a nearly turn-key manner.

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation Navigator and Design Hubs

Xilinx Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select **Help** > **Documentation and Tutorials**.
- On Windows, select **Start** > **All Programs** > **Xilinx Design Tools** > **DocNav**.
- At the Linux command prompt, enter: `docnav`

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

References

1. *Introduction to FPGA Design with Vivado® High-Level Synthesis* (UG998)
2. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
3. *Vivado Design Suite Tutorial: High-Level Synthesis* (UG871)
4. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing* (UG973)
5. *Vivado Design Suite Tutorial: Creating and Packaging Custom IP* (UG1119)
6. *Vivado Design Suite User Guide: Model-Based DSP Design Using System Generator* (UG897)
7. *UltraFast Design Methodology Guide for the Vivado Design Suite* (UG949)
8. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
9. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
10. *Methods for Integrating AXI4-based IP Using Vivado IP Integrator* (XAPP1204)
11. Vivado Design Suite Documentation
12. Xilinx Intellectual Property Page

Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. C-based Design: High-Level Synthesis with the Vivado HLS Tool Training Course
2. C-based HLS Coding for Hardware Designers Training Course
3. C-based HLS Coding for Software Designers Training Course
4. Vivado Design Suite QuickTake Video Tutorials
5. Vivado Design Suite QuickTake Video: Vivado High-Level Synthesis
6. Vivado Design Suite QuickTake Video: Getting Started with High-Level Synthesis
7. Vivado Design Suite QuickTake Video: Verifying your Vivado HLS Design
8. Vivado Design Suite QuickTake Video: Creating Different Types of Projects

Please Read: Important Legal Notices