RapidWright Documentation

Release 2018.2.5-beta

Xilinx Research Labs

Nov 28, 2018

CONTENTS

CHAPTER

ONE

INTRODUCTION

Table of Contents

1.1 What is RapidWright?

RapidWright is an open source Java framework that enables netlist and implementation manipulation of modern Xilinx FPGA and SoC designs. It compliments Xilinx's Vivado® Design Suite and provides developers with capabilities such as:

- Fast loading accurate device model views for all Vivado-supported Xilinx devices (Series 7, UltraScale™, and UltraScale+™)
- Reads and writes unencrypted Vivado Design Checkpoint files (.dcp)
- Hundreds of APIs to help build customized solutions to a wide variety of implementation challenges
- Examples of how to pre-implement (pre-place and pre-route) IP, relocate such blocks and compose pre-implemented blocks together

Note: RapidWright is not an official product from Xilinx and designs created or derived from it are not warranted. Please see LICENSE.TXT for full details.

1.2 Why RapidWright?

We believe that when people are empowered to create tailored solutions to their own specific challenges, innovation takes place. We are building RapidWright to be an environment that fosters this caliber of innovation. The commercial FPGA CAD world is in the unfortunate state of being closed source. We hope that with the release and continued development of RapidWright, we can change the status quo of how we develop and interact with FPGAs.

RapidWright's mission is to:

- Facilitate rapid creation of custom design implementation solutions for FPGAs
- Foster an ecosystem of research and development in academia and industry
- Be fast, efficient, light-weight and easy-to-use
- Serve as a platform that can grow into an open source FPGA implementation flow (future work)

1.3 What about RapidSmith?

RapidWright is a next generation RapidSmith. Previously, RapidSmith was created to enable FPGA CAD tool creation for older Xilinx devices, specifically those supported under ISE. RapidSmith is dependent on the Xilinx Design Language (XDL) which was discontinued in Vivado. Therefore, RapidSmith doesn't work with newer devices supported exclusively in Vivado (although some valiant efforts have been made to bridge the gap[1],[2]).

RapidWright has been significantly overhauled from its parent RapidSmith code. The FPGA device model is cleaner, more data rich, is faster, more memory efficient and adds several insights and capabilities from the Vivado design paradigm. A distinguishing and enabling capability of RapidWright is its ability to read and write unencrypted Vivado Design Checkpoint files. It also maintains full representation of both the logical and physical netlist of FPGA designs.

1.4 Vivado and RapidWright

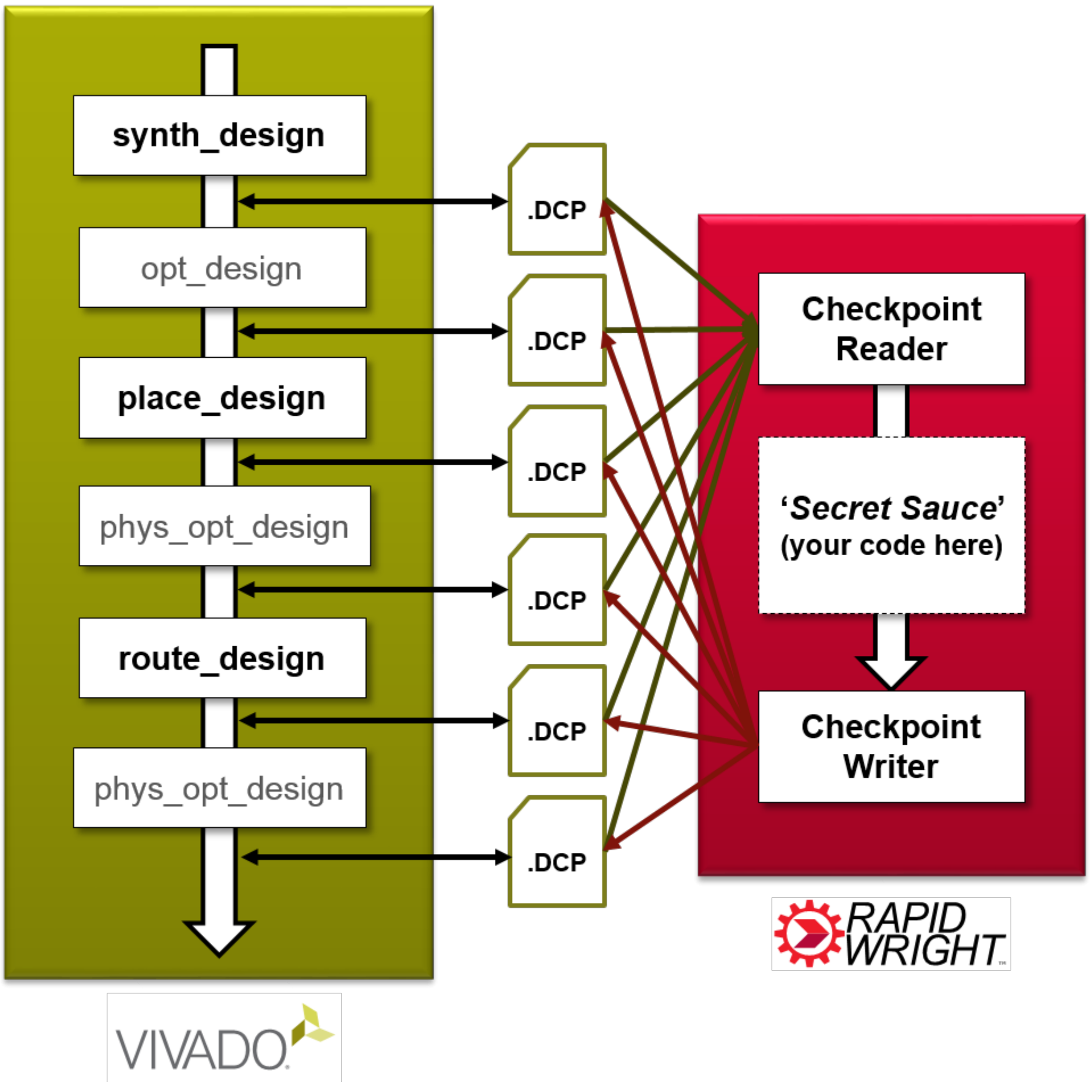
The Vivado Design Suite is the tool environment for developing and implementing designs for Xilinx FPGAs and SoCs. Vivado provides both a GUI environment and a Tcl scripting interface to control the various tools and steps involved in development. The Tcl scripting interface is quite powerful in that it provides users with hundreds of commands to manipulate their design. However, despite the breadth of functionality that the Tcl interface offers, it does have some shortcomings.

- First, some tasks that a user would want to complete using Tcl constructs and commands takes an inordinate amount of runtime making the task infeasible, especially for large designs. For example, attempting to import routing information via Tcl commands for a full design can take several hours or days.
- Second, constructing large, complex operations out of Tcl commands can be inefficient due to its interpreted nature. Many users would also prefer a more mainstream object oriented language with wider support for developing solutions.
- Lastly, if the user wants a particular capability that is not available in the provided library of Tcl commands in Vivado, there is generally no alternative.

RapidWright addresses these shortcomings by providing a means to import, modify and export Vivado-based designs independent of the Tcl interface. It achieves this capability by providing APIs that can read and write design checkpoint files (Vivado's design file format) into and out of the RapidWright framework as illustrated below.

[1] White, Brad S., "Tincr: Integrating Custom CAD Tool Frameworks with the Xilinx Vivado Design Suite" (2014). All Theses and Dissertations. 4338. http://scholarsarchive.byu.edu/etd/4338

[2] Townsend, Thomas James, "Vivado Design Interface: Enabling CAD-Tool Design for Next Generation Xilinx FPGA Devices" (2017). All Theses and Dissertations. 6492. http://scholarsarchive.byu.edu/etd/6492



RapidWright includes a compact, fast-loading device model and hundreds of APIs to help manipulate implementations. These capabilities will enable users to develop new implementation strategies and capabilities that have not been available previously in Vivado. We believe RapidWright provides a foundational framework that opens the door for innovation in the FPGA CAD space.

CHAPTER

TWO

GETTING STARTED

How would you like to use RapidWright?

2.1 Quick Start (Try it out)

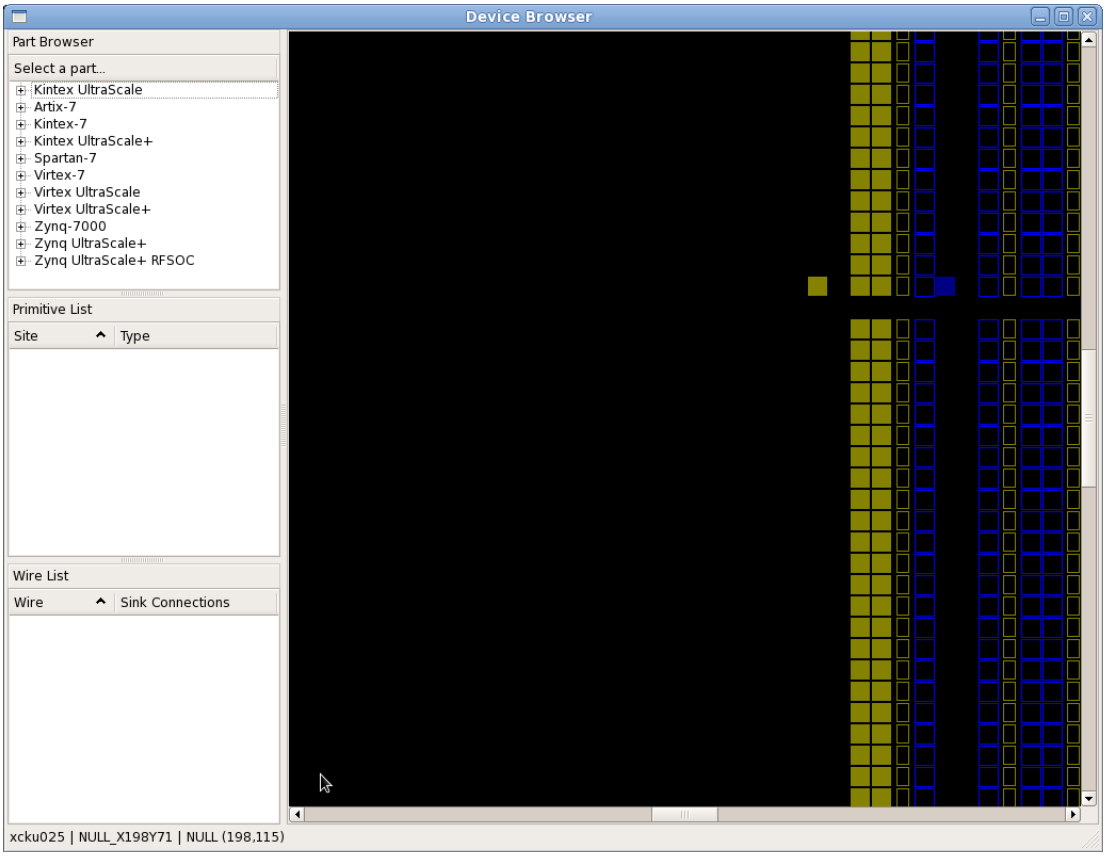
1. Download and install Oracle JRE/JDK Java 1.8 or later (install instructions here)
2. Download the latest standalone RapidWright release jar file
3. Start the RapidWright Python (Jython) interpreter by running:

```
$ java -jar rapidwright-2018.2-standalone-lin64.jar # (or whichever jar you
↪downloaded)
```

At this point you should have a Python interpreter instance running with most RapidWright classes loaded. You can test your install by running the following at the prompt:

```
>>> DeviceBrowser.main([])
```

You should see the GUI come up similar to this screenshot:



If you have gotten to this point, congrats! Your RapidWright install is correctly configured and you are ready to start experimenting.

Note that the standalone jar comes with only a very few select devices:

- AWS-F1: Virtex UltraScale+ VU9P (xcvu9p)
- PYNQ-Z1: Zynq 7020 (xc7z020)
- Virtex UltraScale VU440 (xcvu440)

If you would like to add additional devices, please follow the full setup process below.

2.2 Full Installation (Development)

RapidWright is written in Java and should be able to run on most platforms. However, we currently only test on RedHat 6 or Windows 7 (64-bit).

Pre-requisites

1. Oracle JDK Java 1.8 or later (install instructions here)
2. Git source code revision control system
3. Vivado Design Suite 2018.2 (Not essential to run RapidWright, but makes it useful)
4. (Recommended) An IDE such as Eclipse
5. (Optional) Gradle 4.0 or later (build tool)

2.2.1 Automatic Installer

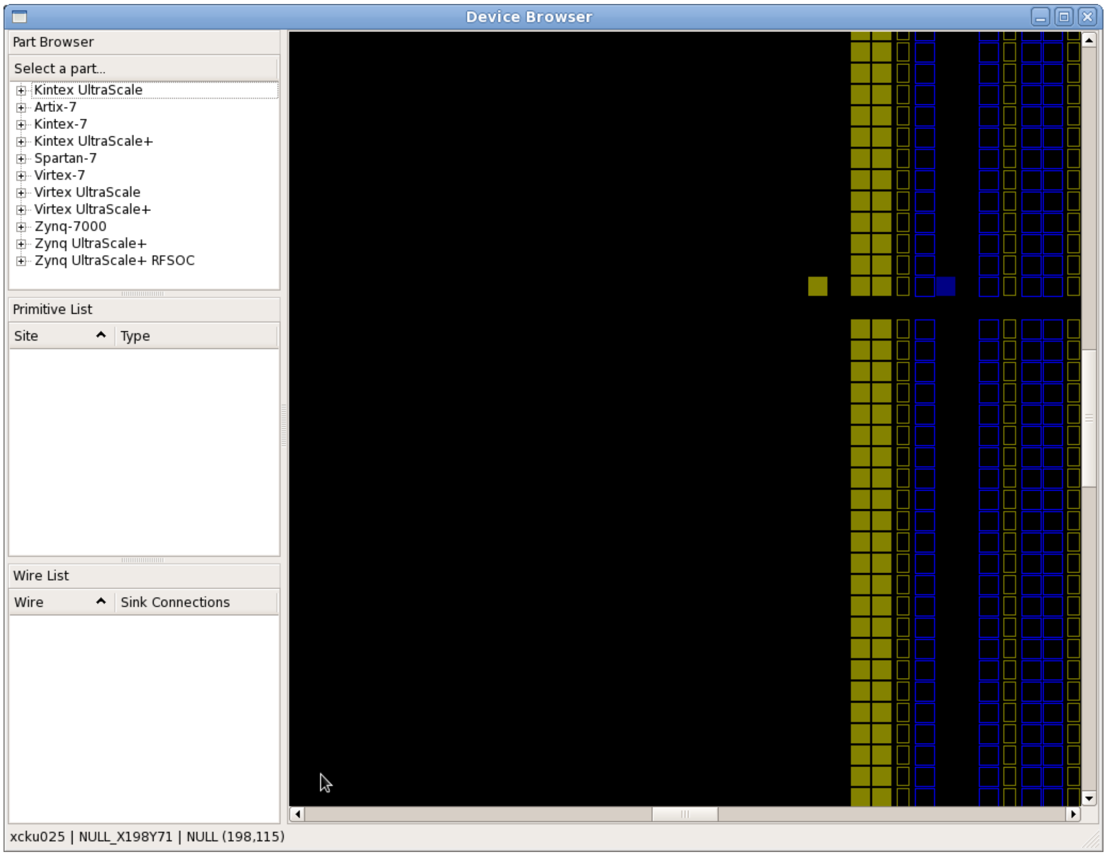
The easiest way to get RapidWright setup is to use the automatic installer jar that performs the manual installation automatically. Make sure you have the JDK and Git on your PATH.

1. Download `rapidwright-installer.jar` to the directory where you would like RapidWright to reside.
2. From a terminal in that directory, run `java -jar rapidwright-installer.jar` (To open a terminal on Windows, search and run 'cmd.exe' from the Start orb)
3. Use one of the BASH/CSH/BAT scripts created at the end of the install to set the proper environment variables for subsequent invocations of RapidWright.
4. (Optional) You can setup Eclipse after the automatic install by following *Setup Eclipse with Existing Repo*.

2.2.2 Manual Installation Steps

RapidWright source code and data files are hosted on GitHub. Here is how to get the necessary files to get started:

1. Use `git clone https://github.com/Xilinx/RapidWright.git` to clone the repo, either on the command line or setting up a new project in your IDE. For a detailed tutorial setting up RapidWright in Eclipse see the *RapidWright Eclipse Setup* page.
2. Go to https://github.com/Xilinx/RapidWright/releases and download the latest release files: rapidwright_data.zip and rapidwright_jars.zip.
3. Expand the two zip files into the root repository directory, there should be a 'jars' and 'data' directory listed there. Make sure to delete previous 'jars' and 'data' directories if present.
4. Set the environment variable `RAPIDWRIGHT_PATH=<your_repo_path>`
5. Be sure to add the compiled Java files and jar files in the jar folder to your `CLASSPATH` variable. If using Bash and can delete the unused OS-specific jars in the jars directory, you could add the following to your `.bashrc` file: `export CLASSPATH=$RAPIDWRIGHT_PATH:$(echo $RAPIDWRIGHT_PATH/jars/*.jar | tr ' ' ':')`
6. Compile the project either through an IDE (such as Eclipse, etc). You may need to refresh the project to ensure the IDE can see the jars added in step 3. You can also use Gradle to compile the project using the provided gradle build script. You will need to make sure Gradle is installed and then run: `gradle build -p $RAPIDWRIGHT_PATH`
7. A quick test is to try running the DeviceBrowser class with something like: `java com.xilinx.rapidwright.device.browser.DeviceBrowser`. You should see the GUI come up similar to this screenshot:



If you have gotten to this point, congrats! Your RapidWright install is correctly configured and you are ready to start experimenting.

At this point if you are familiar enough with FPGAs, Xilinx architecture and nomenclature, feel free to skip to the *RapidWright Overview* section, otherwise read on.

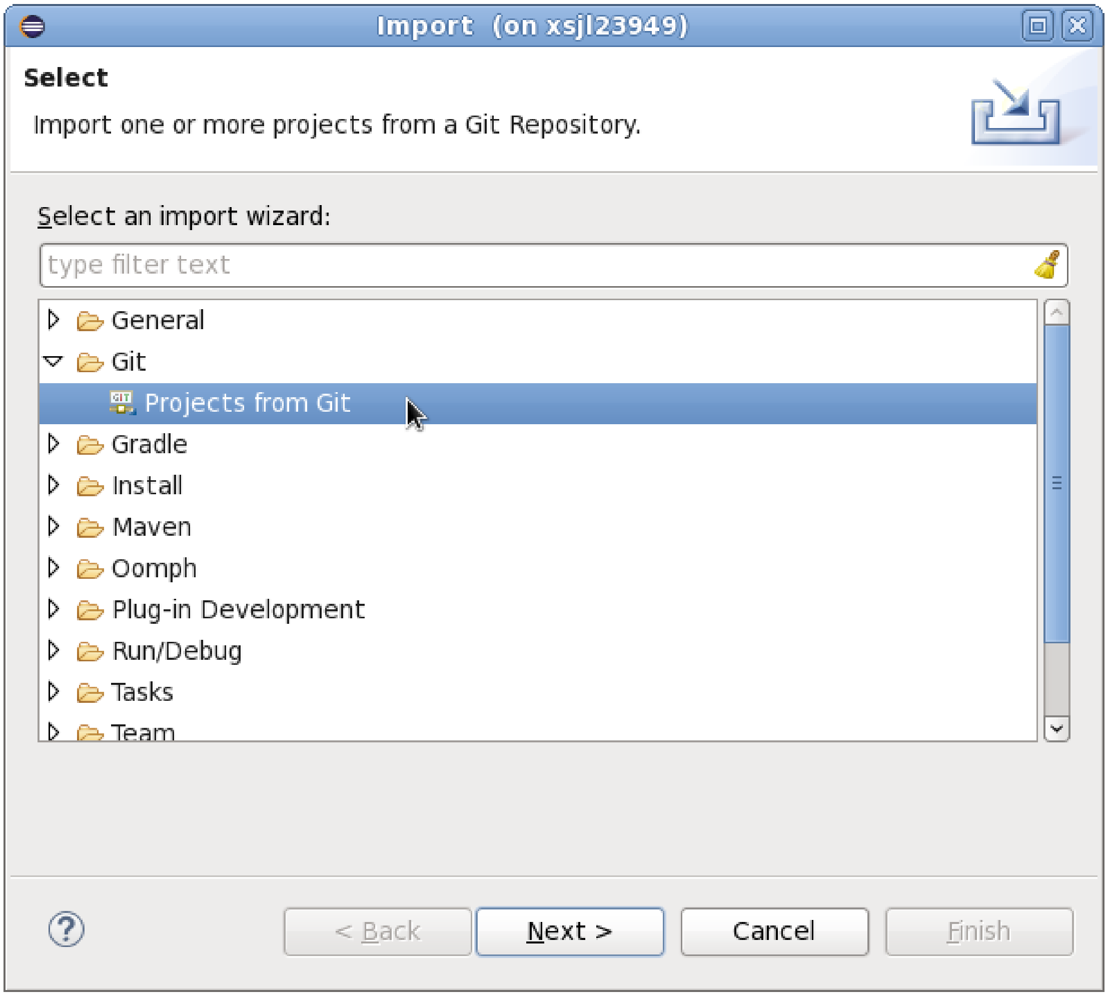
CHAPTER

THREE

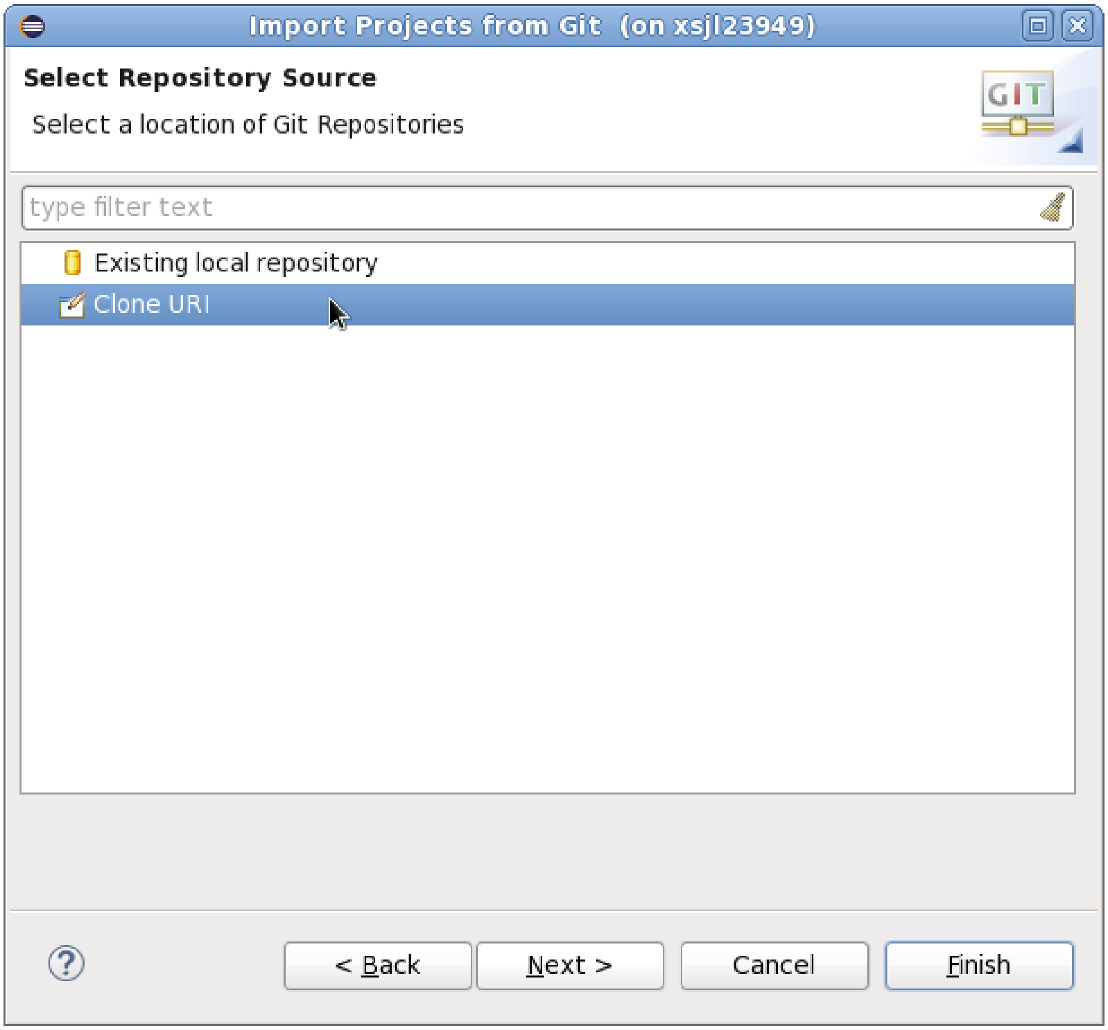
RAPIDWRIGHT ECLIPSE SETUP

3.1 Step-by-Step Instructions

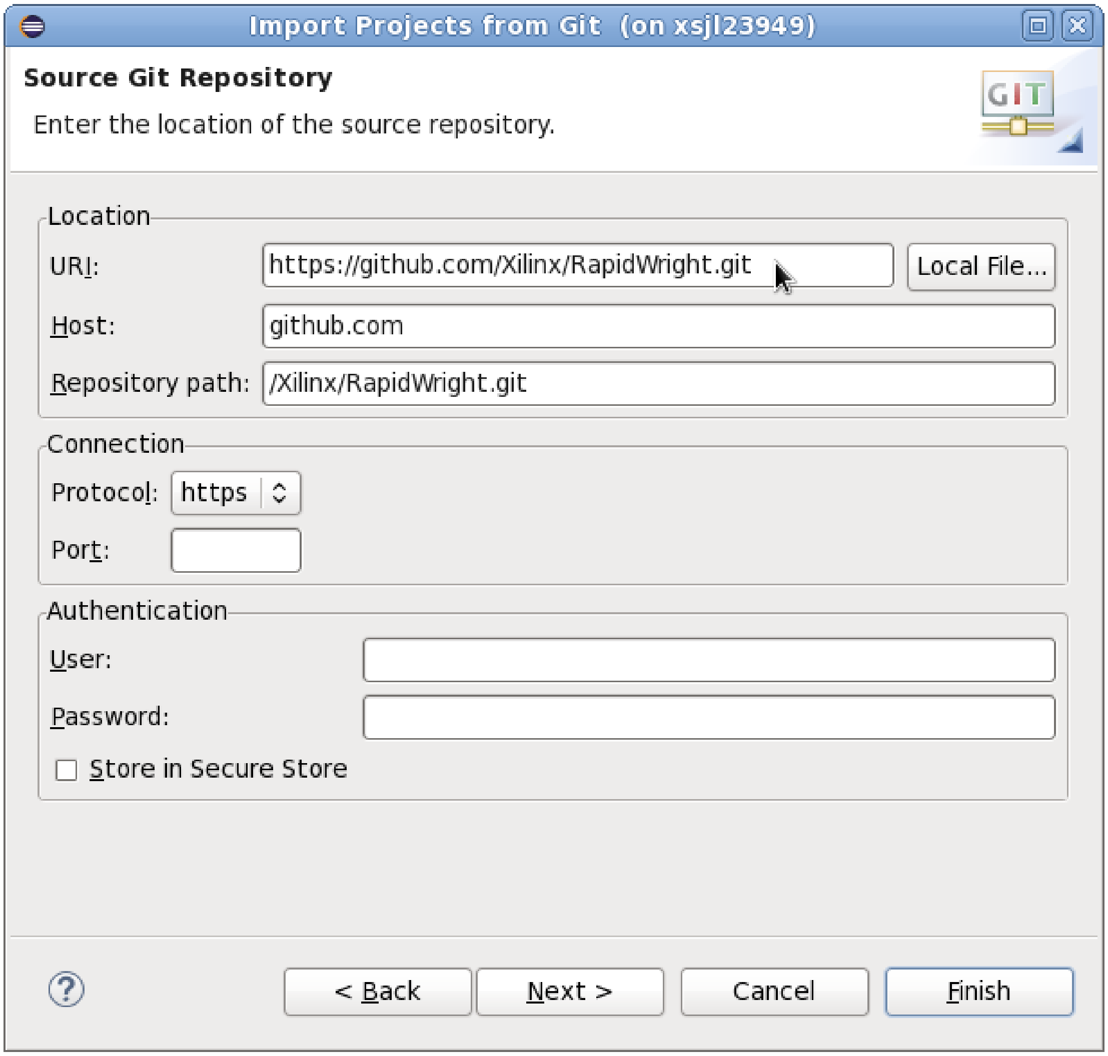
1. Make sure you have Java JDK 1.8 (or later) installed: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html Follow the instructions when running the downloaded executable. Add the `$(YOUR_JDK_INSTALL_LOCATION)/jdk1.x.x_x/bin` folder to your `PATH` environment variable.
2. Download Eclipse: http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygen2
3. Install Eclipse by extracting the archive into a desired folder on your computer
4. Run Eclipse (you may want to add the executable to your path)
5. In Eclipse, choose the File->Import… menu option. This will bring up a dialog, choose the Git/Projects from Git option as shown in the screenshot below (click Next):



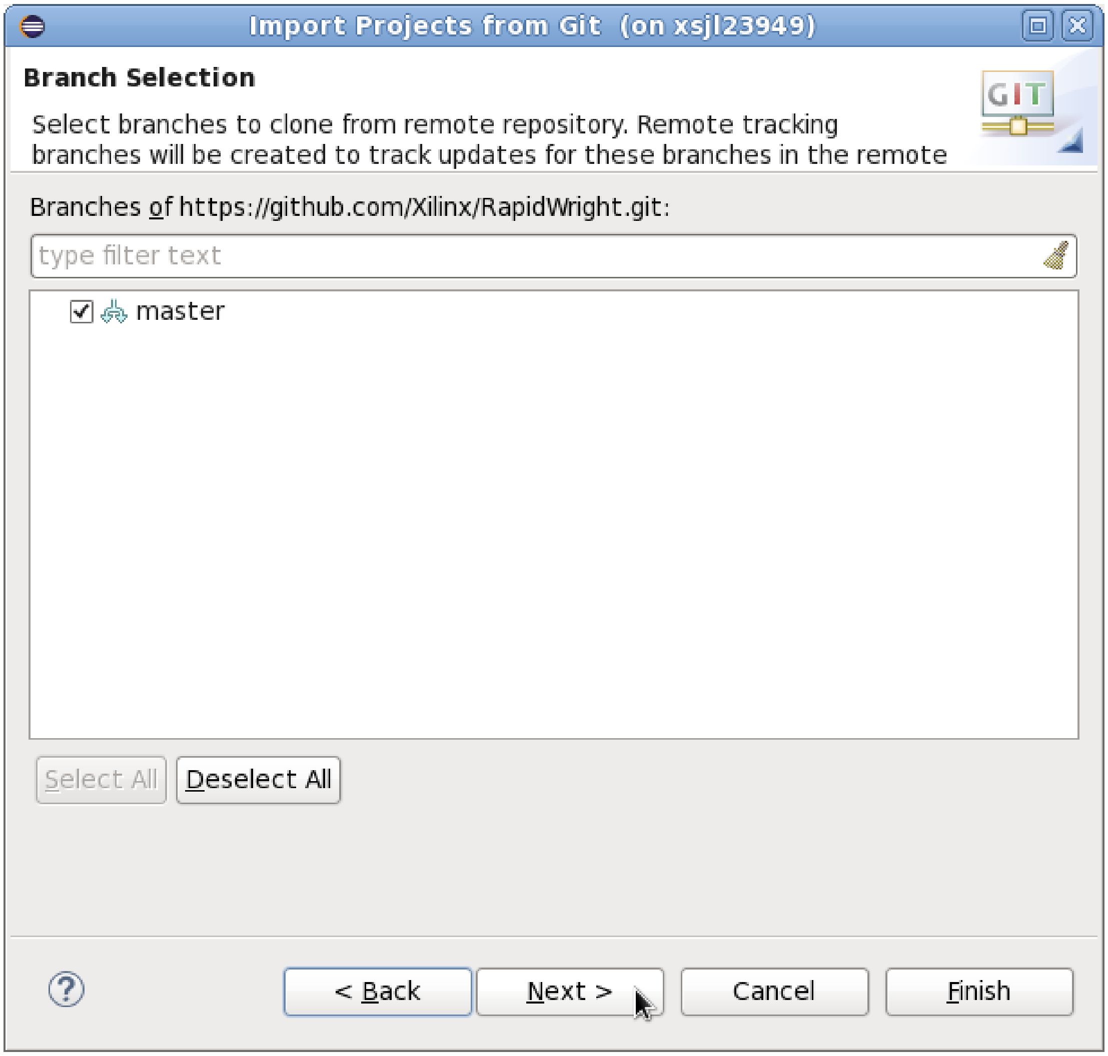
6. Choose Clone URI and click Next:



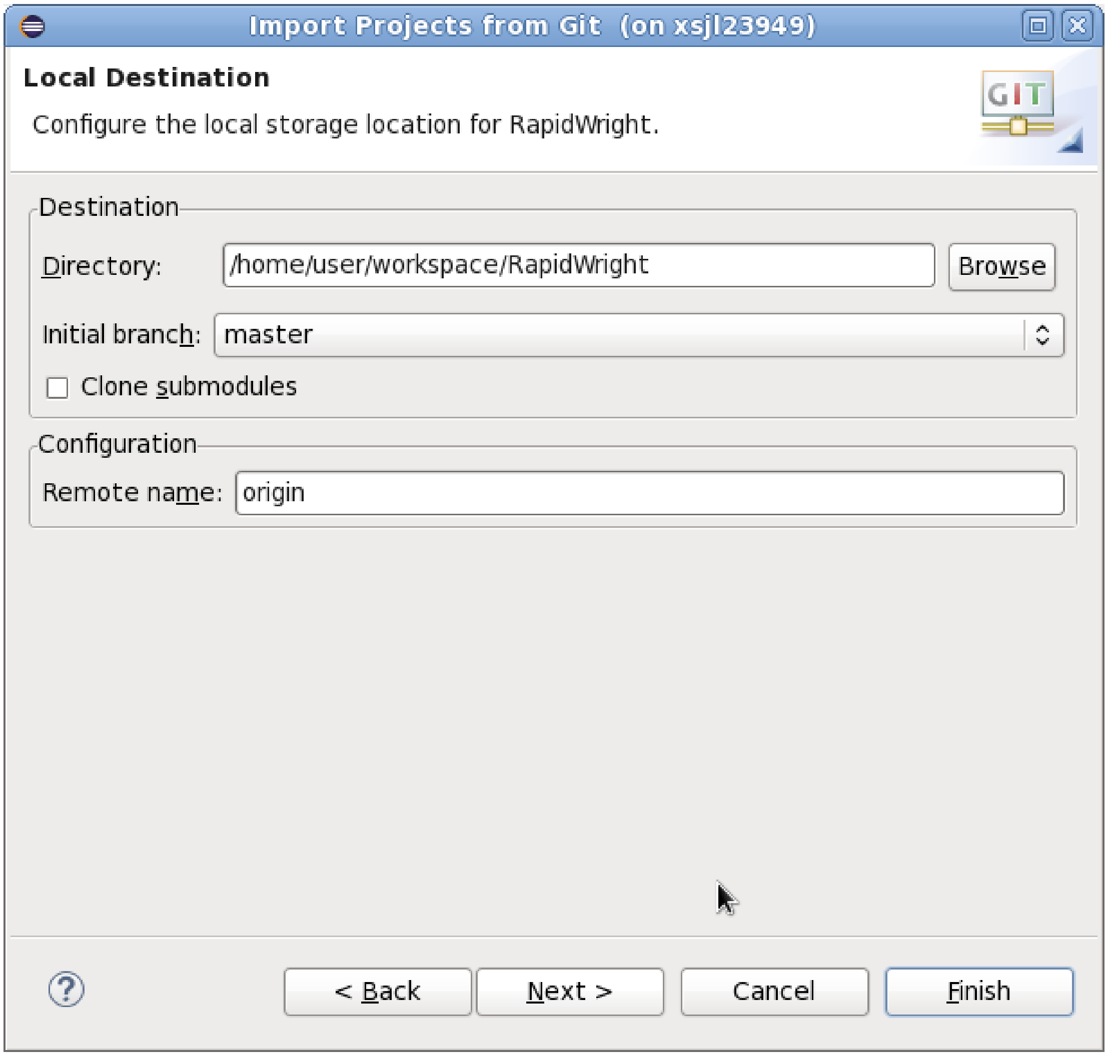
7. Copy and paste `https://github.com/Xilinx/RapidWright.git` into the URI box as shown below. The Host and Repository path fields should automatically be populated. Enter user and password (if applicable).



8. Choose the master branch, click next:



9. Choose the location of where you want Eclipse to put your RapidWright workspace. Preferably, you should choose a workspace directory with any other Eclipse projects such as `/home/user/workspace/RapidWright`. Click next to have Eclipse clone the repo into your workspace.

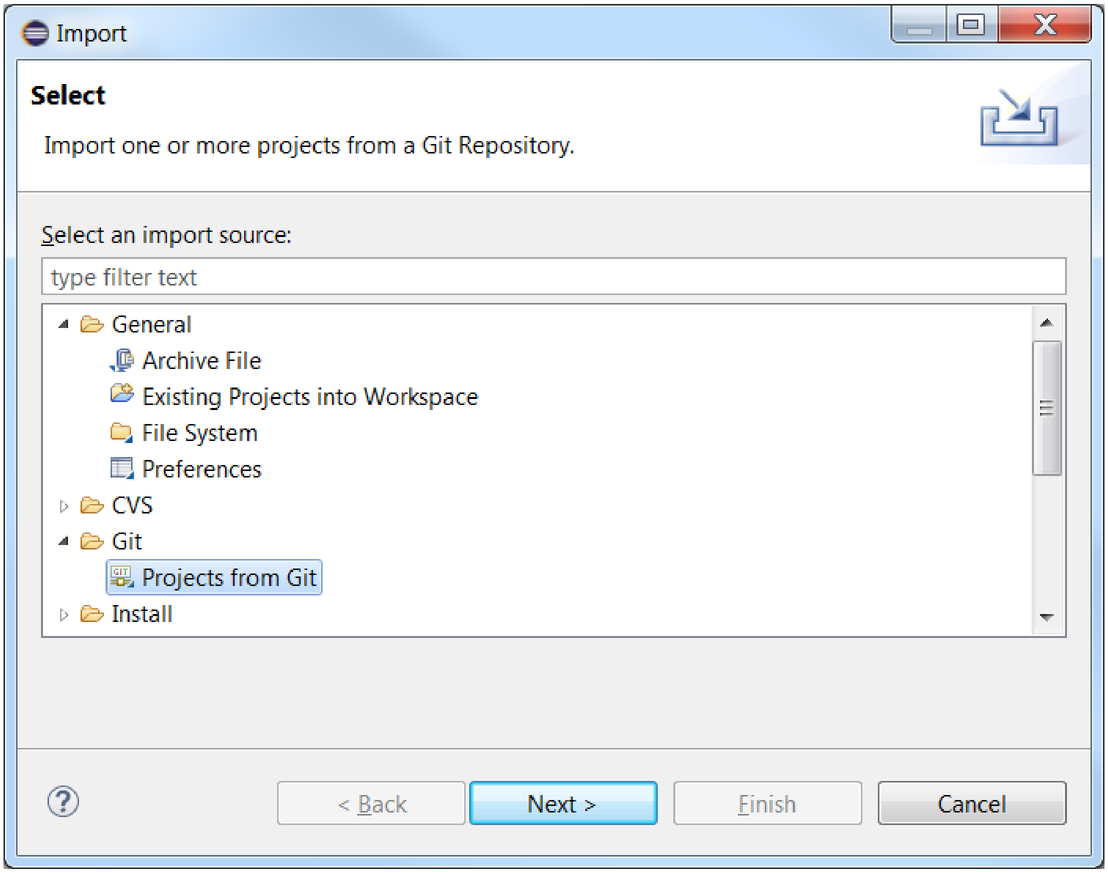


10. Continue with step 2 back on the *Manual Installation Steps* section of the *Getting Started* page.

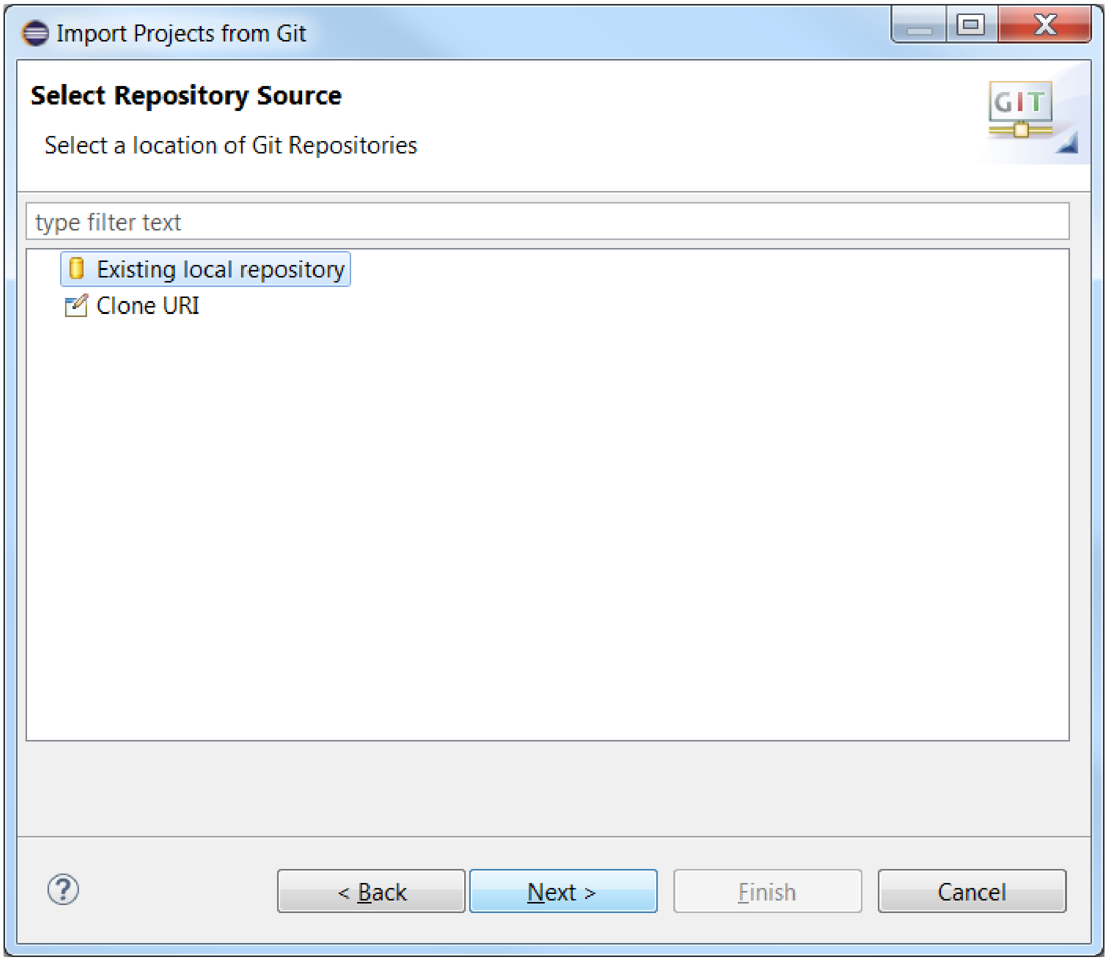
3.2 Setup Eclipse with Existing Repo

If you already have the RapidWright repository checked out, you can import it into an Eclipse workspace by following these steps (you can skip to Step 5 if you already have Eclipse installed and open)

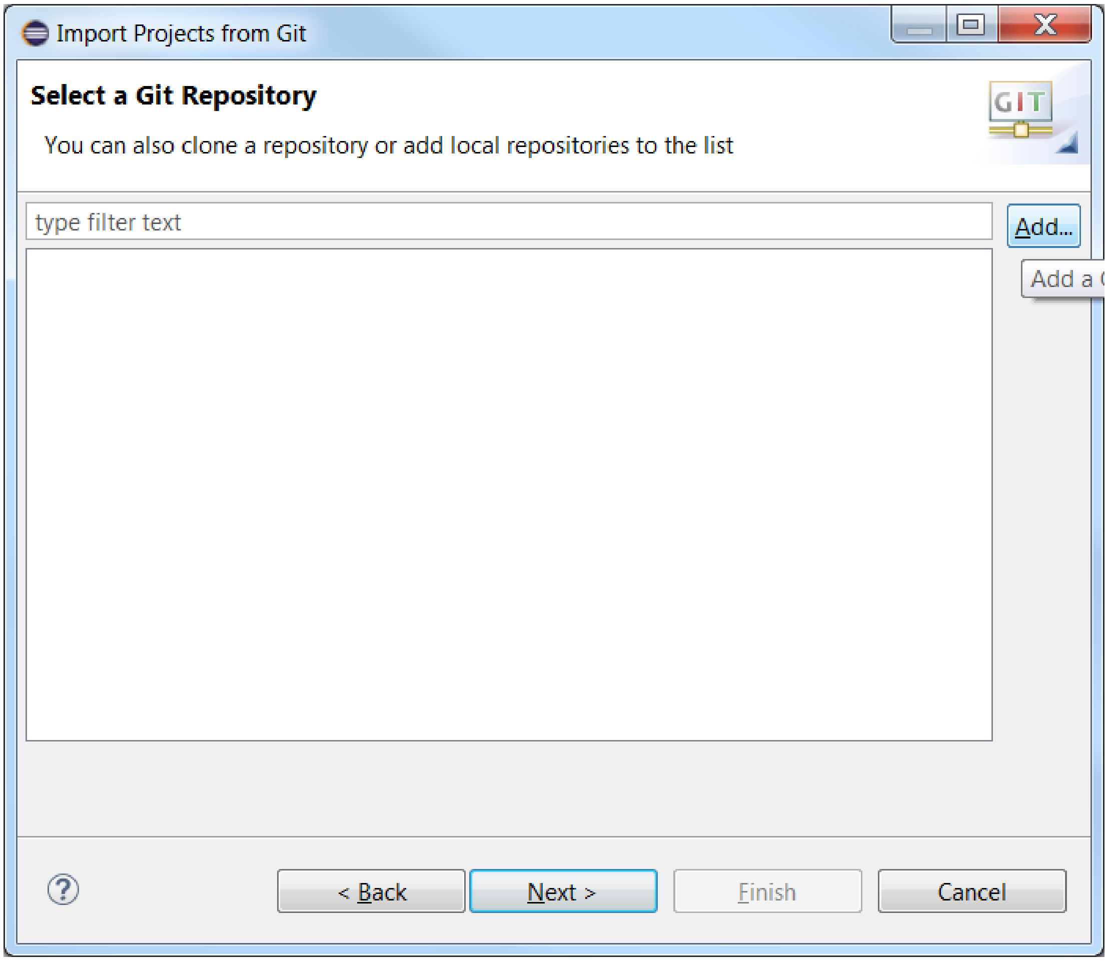
1. Make sure you have Java JDK 1.8 (or later) installed: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html Follow the instructions when running the downloaded executable. Add the `$(YOUR_JDK_INSTALL_LOCATION)/jdk1.x.x_x/bin` folder to your `PATH` environment variable.
2. Download Eclipse: http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/oxygen2
3. Install Eclipse by extracting the archive into a desired folder on your computer
4. Run Eclipse (you may want to add the executable to your path)
5. In Eclipse, choose the File->Import… menu option. This will bring up a dialog, choose the Git/Projects from Git option as shown in the screenshot below (click Next):



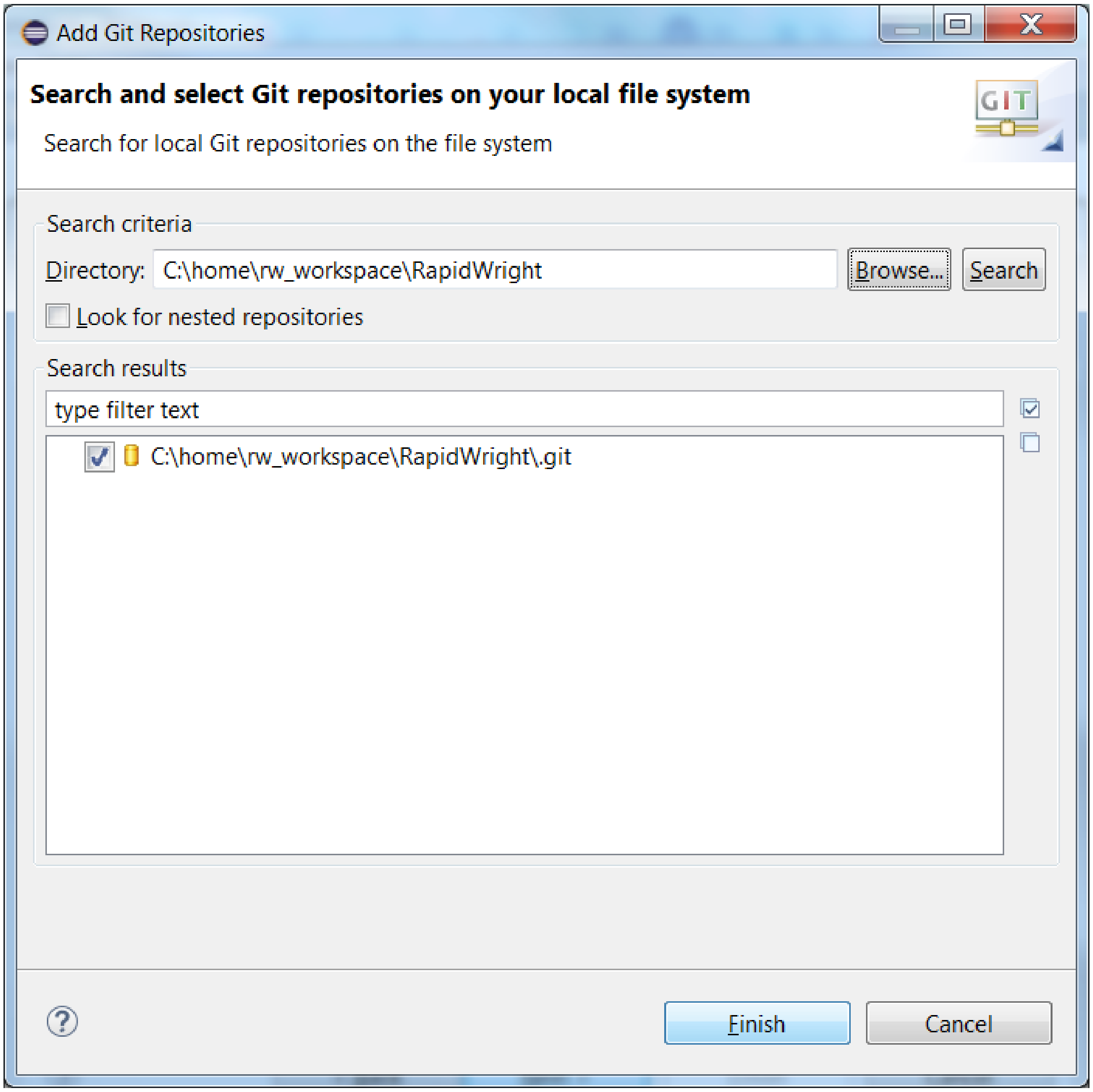
6. Choose 'Existing local repository', then click Next



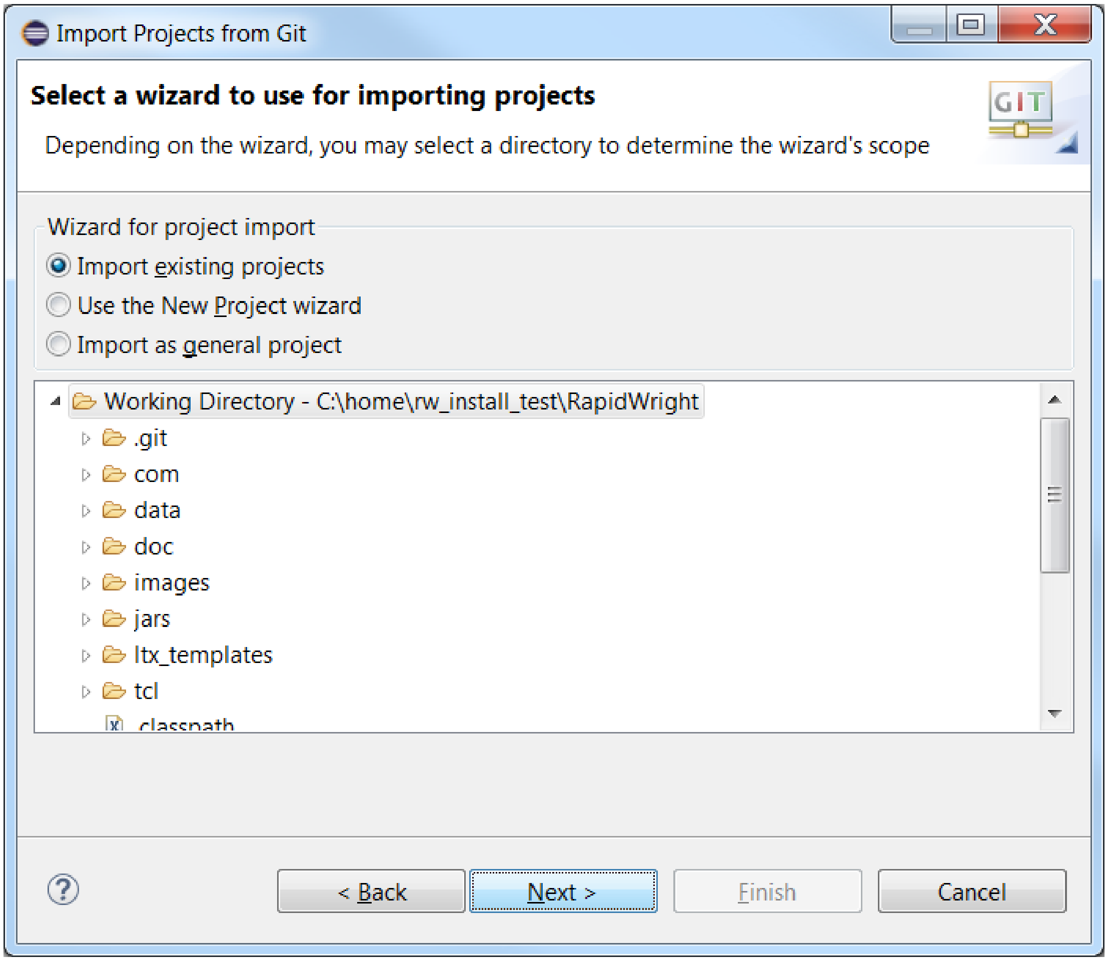
7. Select the existing repository by clicking the 'Add…' button



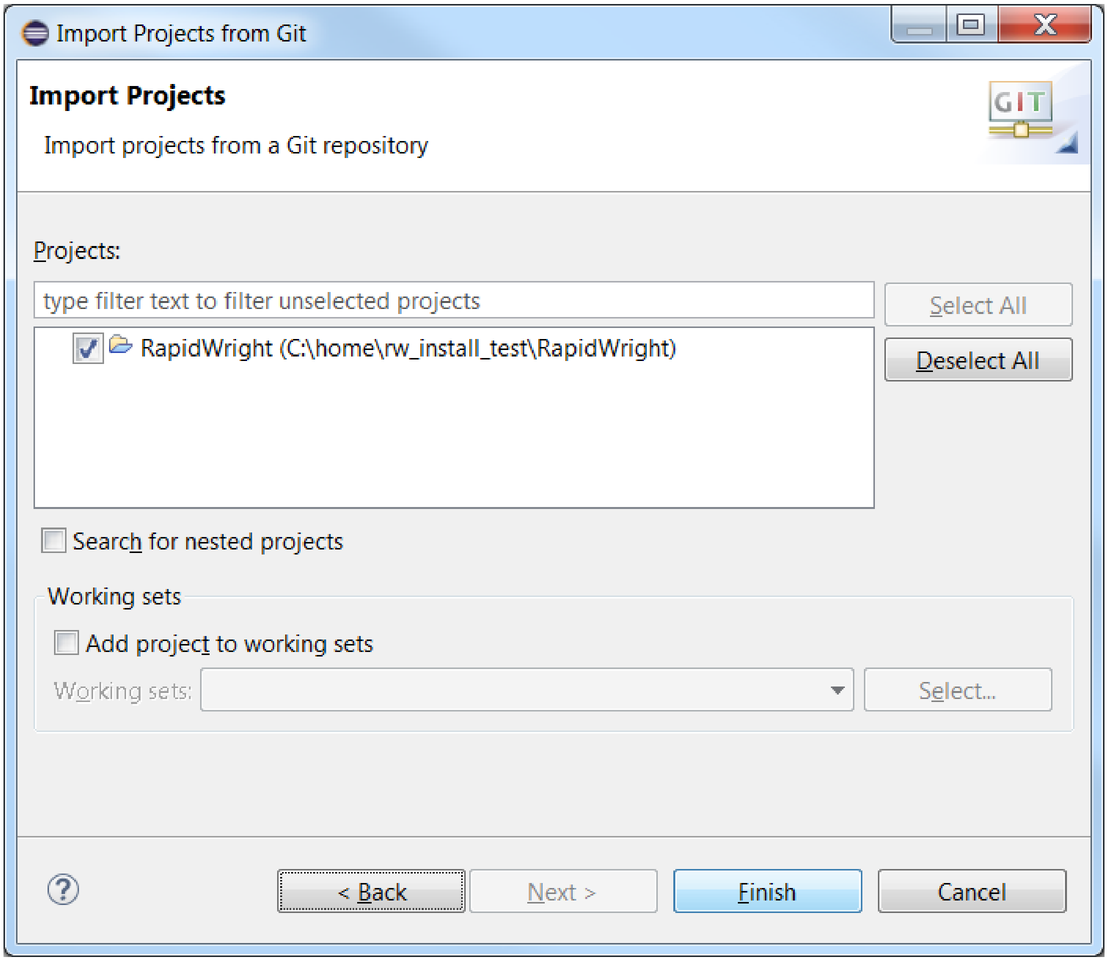
8. Enter the location of the repository in the 'Directory:' text box, check the box next to the name of the repo once it appears in the lower window. Click 'Finish' and then 'Next' on the previous window.



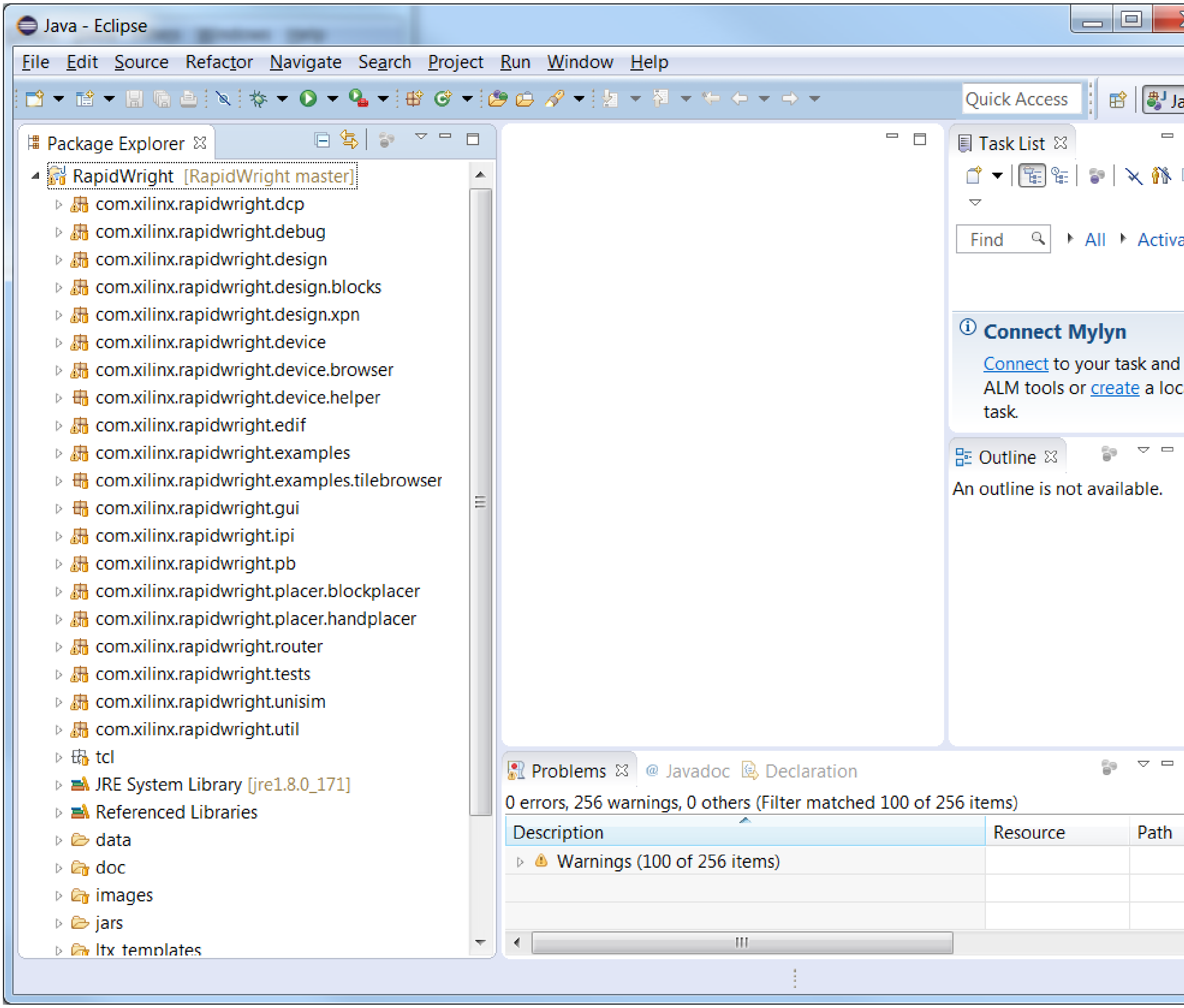
9. On the Wizard selection window, choose 'Import existing projects'. Then, click 'Next'.



10. Finally, click 'Finish' to finalize the import.



11. Eclipse will then import the project, compile all the source and it should look similar to the screenshot below:


Java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Package Explorer
RapidWright [RapidWright master]
com.xilinx.rapidwright.dcp
com.xilinx.rapidwright.debug
com.xilinx.rapidwright.design
com.xilinx.rapidwright.design.blocks
com.xilinx.rapidwright.design.xpn
com.xilinx.rapidwright.device
com.xilinx.rapidwright.device.browser
com.xilinx.rapidwright.device.helper
com.xilinx.rapidwright.edif
com.xilinx.rapidwright.examples
com.xilinx.rapidwright.examples.tilebrowser
com.xilinx.rapidwright.gui
com.xilinx.rapidwright.ipi
com.xilinx.rapidwright.pb
com.xilinx.rapidwright.placer.blockplacer
com.xilinx.rapidwright.placer.handplacer
com.xilinx.rapidwright.router
com.xilinx.rapidwright.tests
com.xilinx.rapidwright.unisim
com.xilinx.rapidwright.util
tcl
JRE System Library [jre1.8.0_171]
Referenced Libraries
data
doc
images
jars
Task List
Find
All
Connect Mylyn
Connect to your task and
ALM tools or create a loc
task.
Outline
An outline is not available.
Problems
Javadoc
Declaration
0 errors, 256 warnings, 0 others (Filter matched 100 of 256 items)
Description
Resource
Path
Warnings (100 of 256 items)

CHAPTER

FOUR

FPGA ARCHITECTURE BASICS

Table of Contents

This section is meant as a brief introduction to FPGA architecture and technology. Most people familiar with FPGAs can easily skip this section.

4.1 What is an FPGA?

An field programmable gate array (FPGA) is a special kind of chip (integrated circuit, silicon device, microchip, computer chip, or whatever designation is most familiar) that can be programmed to behave essentially like any other chip. One might think that a microprocessor or CPU falls into such a description as it is programmable through software compilation. However, an FPGA and CPU differ significantly in architecture and programming model.

4.2 CPU vs. FPGA

A central processing unit (CPU or just processor) follows the Von Neumann compute-based architecture as illustrated in the figure below.

A control unit driven by instructions fetched from memory drives the flow of input data through the processor's registers and logic producing outputs. The data paths, instruction set, register counts and memory interface are all fixed at the time of fabrication of the CPU. That is, they are unchanging attributes of the processor and cannot be customized later.

In stark contrast to the CPU architecture, an FPGA has highly configurable logic and data paths. This is enabled by a bit-wise, fine-grained architectural model to realize computation. In order to better understand how FPGAs work, it is beneficial to comprehend their atomic units of computation. Although modern FPGAs have a wide variety of

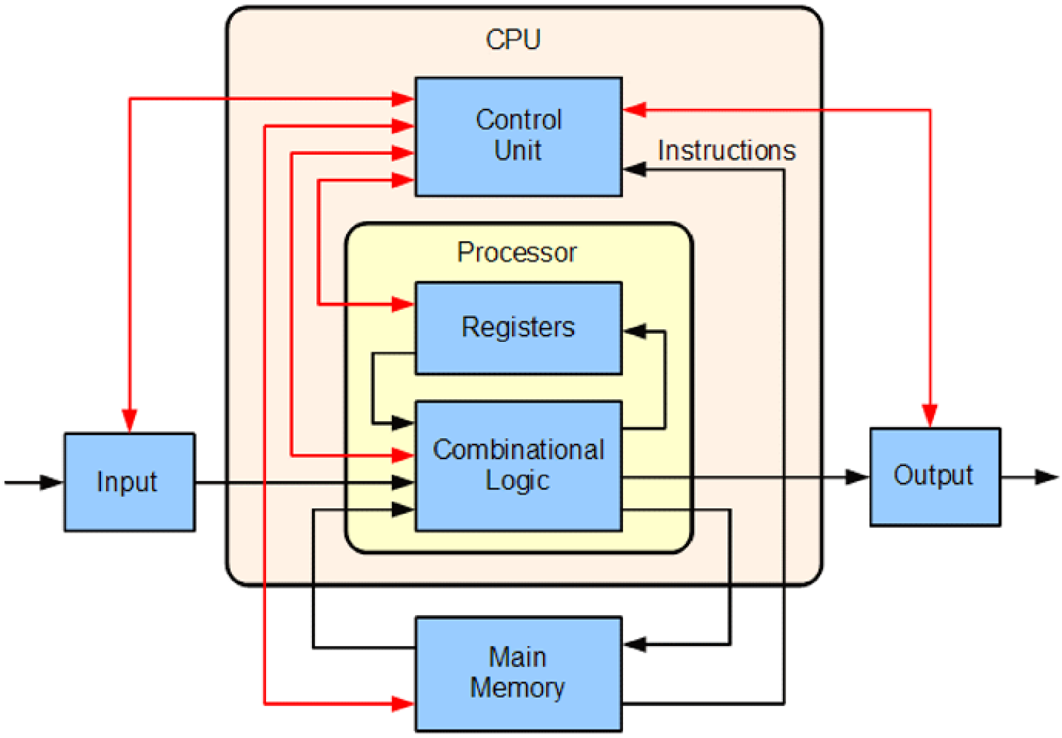


Fig. 4.1: Basic Von Neumann Processing Model for CPUs (Source: Labtron, Creative Commons).

components, at their heart is a large array of replicated programmable look-up tables (LUTs), flip-flops (or registers) and programmable wires called interconnect as seen in the figures below.

4.3 Lookup Tables (LUTs)

At the heart of configurable logic in FPGAs, lies a basic atomic unit of computation, a lookup table or LUT. A LUT has a single bit output that is calculated based on the input signal values and the configurable table (or memory) entries as shown in the figure below.

Although mainstream FPGAs typically use 6-input LUTs, this example illustrates a 3-input LUT for simplicity but the principle of operation is the same.

LUTs are typically constructed using an N:1 multiplexer (shown in green in Figure 4b) and an Nx1-bit memory (shown in blue). The example in the figure above is a LUT where N=8. The number of inputs of a LUT is calculated as the log base 2 of N.

The memory entries in blue boxes in part (b) of the figure above represent the configurable table entries under the 'out' column in part (a). The vector of programming bits {a, b, . . . h} ultimately decide how the LUT will behave given different values presented on the inputs {i0, i1, i2}. For example, to program the LUT to evaluate "i0 XOR i1" on the inputs, the programming vector {a=0,b=1,c=1,d=0,e=0,f=1,g=1,h=0} would be used. A LUT can implement any Boolean logic equation limited only by the number of inputs of the LUT's size. This characteristic is illustrated in the figure below. LUTs are commonly chained or combined in series to implement larger Boolean equations.

In some devices, some of the LUTs have additional functionality then enable them to act as small RAMs. These RAMs can be chained together to build larger RAMs as well.

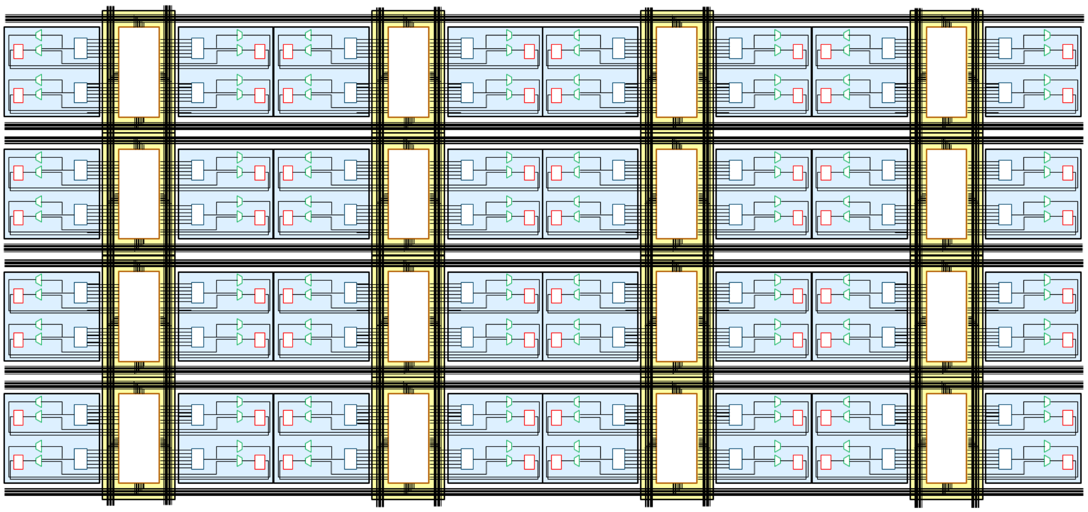


Fig. 4.2: Hypothetical FPGA logic array of LUTs, flip flops and programmable wires (interconnect)

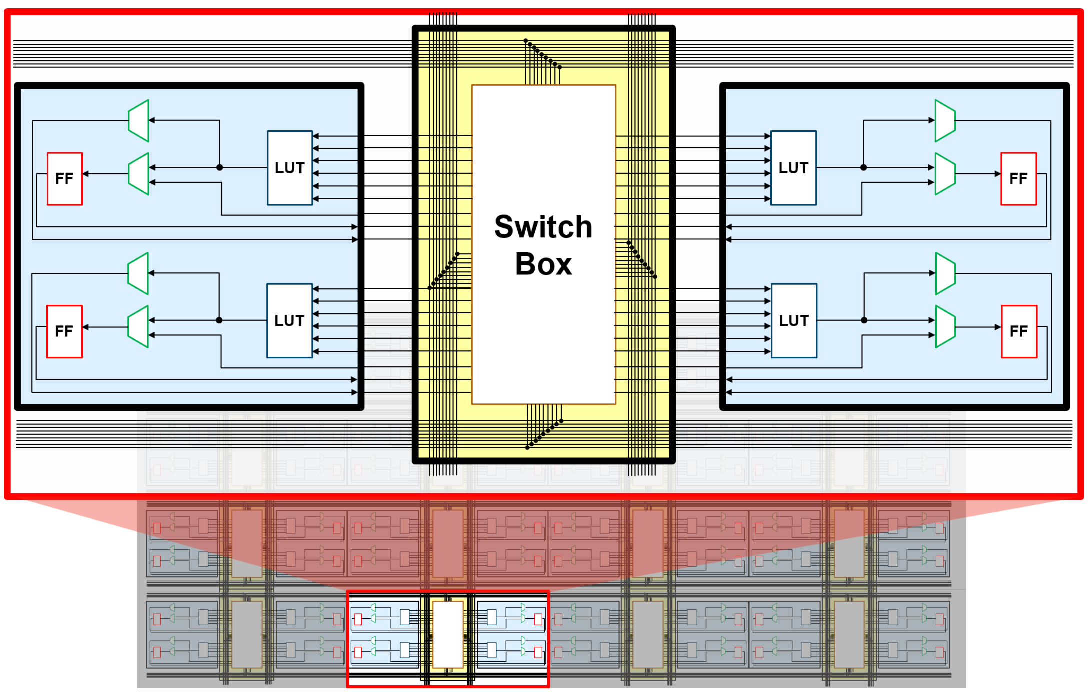


Fig. 4.3: Close up view of replicated tiles of the logic array and interconnect

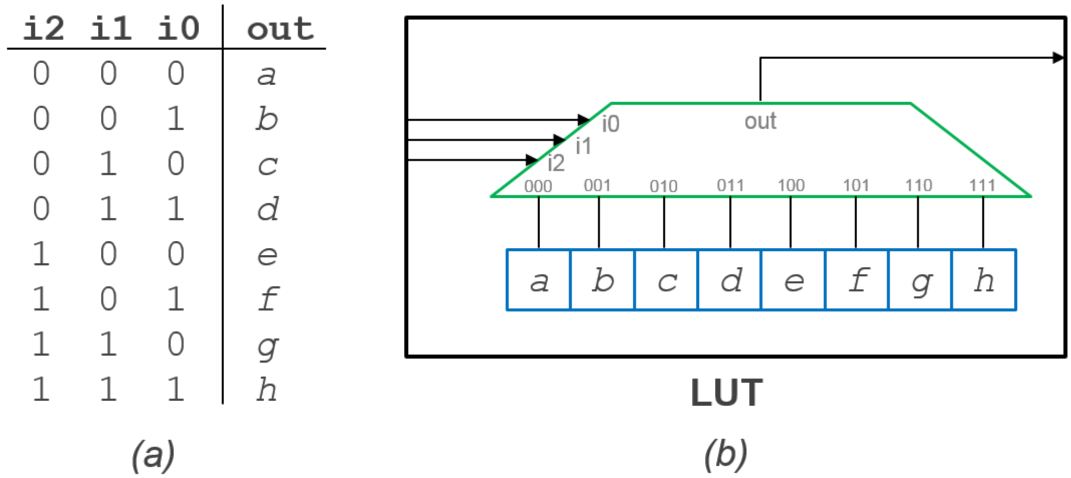


Fig. 4.4: (a) Truth table relationship of a LUT (b) Diagram of logical behaviour of a LUT

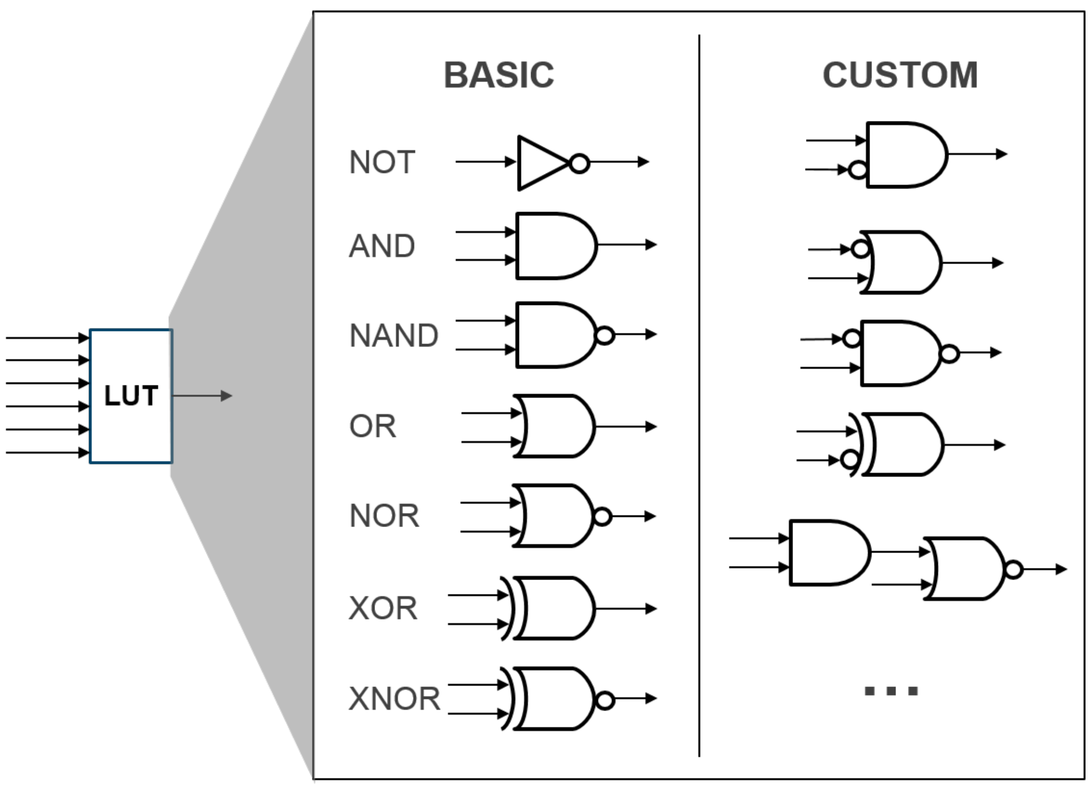


Fig. 4.5: Examples of several (but not all) logic functions a LUT can potentially implement

4.4 State Elements

Once a value is computed from a LUT, it often is desirable to store it. For this purpose, most FPGAs pair their LUTs with a D-flip-flop or equivalent state element. Often the storage element has configurable reset/clear and clock enable signals with an option of making it behave as a latch. These state elements have dedicated clocking paths to help minimize clock skew.

By chaining together LUTs and storing results in flip flops, FPGAs can implement any number of functions and computation limited only by the number of resources of the device and its delay.

Xilinx offers a variant of LUTs that enable them to also store data in the lookup portion of the table such that they can perform as small memories, shifters or FIFOs. More information on this can be found in Series 7 CLB User's Guide or UltraScale CLB User's Guide.

4.5 Carry Chains

Carry chain blocks are primitive elements that are provided with a group of LUTs to enable more efficient programmable arithmetic. Primarily it provides dedicated paths for the carry logic of simple arithmetic operations (add, subtract, comparisons, equals, etc). Implementing these arithmetic operations in LUTs would result in an inefficient use of resources and performance would suffer.

For more detailed information of Xilinx carry chains, please see Series 7 CLB User's Guide or UltraScale CLB User's Guide.

4.6 DSP Blocks

Multiplication on FPGAs can be quite expensive when implemented in LUTs and is a common operation. Therefore, dedicated hard blocks to provide integer multiplication have been present in FPGAs for several years. As applications have evolved, multiplier blocks have evolved to support a variety of DSP-friendly operations such as MAC (multiply, accumulate), wide AND/XOR and several others.

For more detailed information of Xilinx DSP blocks, please see Series 7 DSP User's Guide or UltraScale DSP User's Guide.

4.7 Block RAMs

Larger memories (than those made available as small LUTs) are also a significant resource on FPGAs that generally provide several kilobits of memory storage (Xilinx typically makes 18k or 36k available). These memories are provided in the fabric and are highly configurable and compose-able such that larger memories with several features can be made a available.

For more detailed information of Xilinx Block RAMs, please see Series 7 Memory User's Guide or UltraScale Memory User's Guide

CHAPTER

FIVE

XILINX ARCHITECTURE TERMINOLOGY

Table of Contents

In order to use RapidWright, an understanding of Xilinx FPGA architecture and hierarchy will be necessary in navigating your way around the device APIs. In Xilinx FPGAs, there are six major levels of hierarchy that describe basic components all the way up to the entire device. This hierarchy can be seen in the figure below:

We begin our discussion with a bottom-up approach starting with the lowest level of hierarchy, the basic element of logic.

5.1 BEL (Basic Element of Logic)

At the lowest level, the atomic unit of Xilinx FPGAs is a BEL. BELs are the smallest, indivisible, representable component in the fabric of an FPGA. There are two kinds of BELs, Logic BELs (Basic Element of Logic) and Routing BELs. A Logic BEL is a configurable logic-based site that can support the implementation of a design cell. Each BEL can support one or more types of UNISIM cells (UNISIM cells are described in Libraries Guides UG953 for Series 7 devices and UG974 for UltraScale™ devices). The mapping between a leaf cell (non-leaf cells do not represent implementable hardware, just hierarchy) in the netlist and a BEL site is referred to as the 'placement' of the cell. Thus, when one runs the Vivado command `place_design`, it is essentially mapping all leaf cells in the netlist to compatible and legal BEL sites.

Routing BELs are programmable routing muxes used to route signals between BELs. Routing BELs do not support any design elements (logic cells from the netlist do not occupy routing BEL sites), they are used only for routing. However, some routing BELs do have optional inversions.

BELs have input and output pins. BELs also have configurable connections that connect an input pin to an output pin. These BEL-based configurable connections are called site PIPs (where PIP stands for Programmable Interconnect Point). Both logic BELs and routing BELs can have site PIPs. However, in the case of a logic BEL, the site must be unoccupied by a cell in order for the route through to be usable. Often, these site PIPs, when implemented in logic

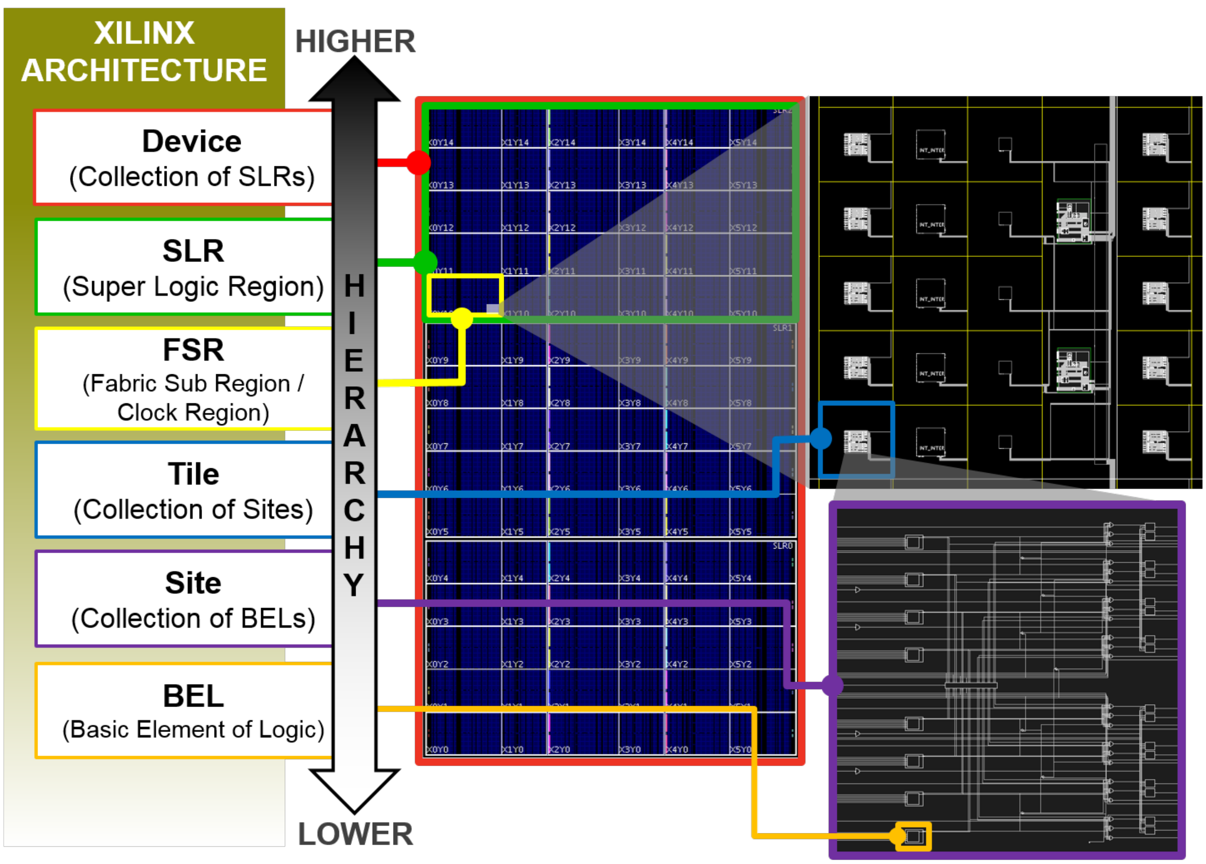


Fig. 5.1: Levels of architectural hierarchy in Xilinx FPGAs.

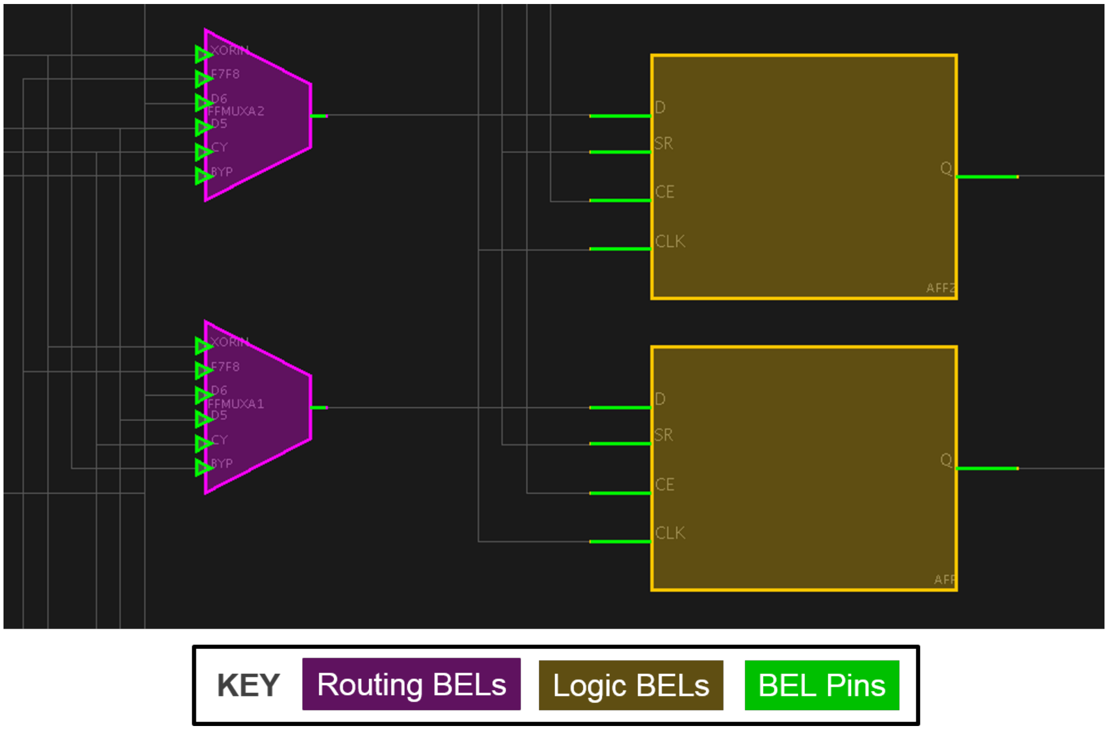


Fig. 5.2: Vivado representation of two routing muxes (routing BELs) and two flip flops (logic BELs).

BELs (a LUT is a common example), are referred to as a "route through" or "route-thru." When routing a design, in order to physically route a net it is sometimes necessary to route through unused LUTs or other logic BELs with site PIPs.

5.2 Site

A group of related elements and their connectivity is referred to as a site. Inside of a site, one can find three major categories of objects:

1. BELs (Logic BELs and/or Routing BELs)
2. Site Pins (External input and output pins to the site)
3. Site wires (connecting elements to each other and site pins)

Sites are instances of a type and each site has a unique name with an `_X#Y#` suffix denoting its location in the site type grid. Each site type will have its own XY coordinate grid, independent of others. The only exception to this is that SLICEL and SLICEM types share the same grid space. SLICEL and SLICEM are the most common site type and are the basic configurable logic building blocks that contain LUTs and flip flops that form the backbone of the FPGA fabric.

5.2.1 Site Type

Sites are heavily replicated across the device and each instance of a site corresponds to a site type of that device's architecture family. Additionally, sites found in an FPGA device are sometimes capable of hosting different types, however, when a tile is queried, a 'primary' site type is designated.

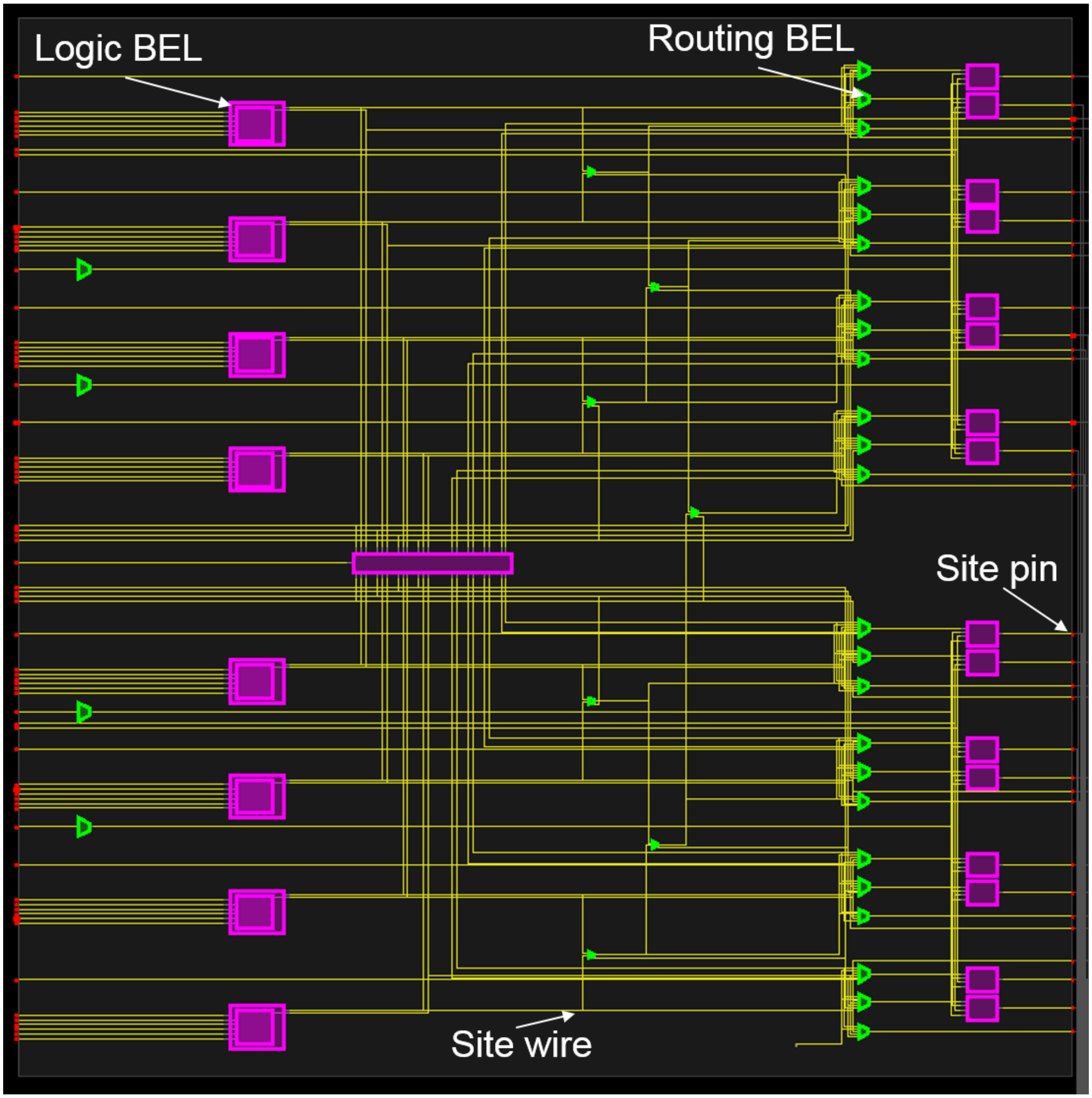
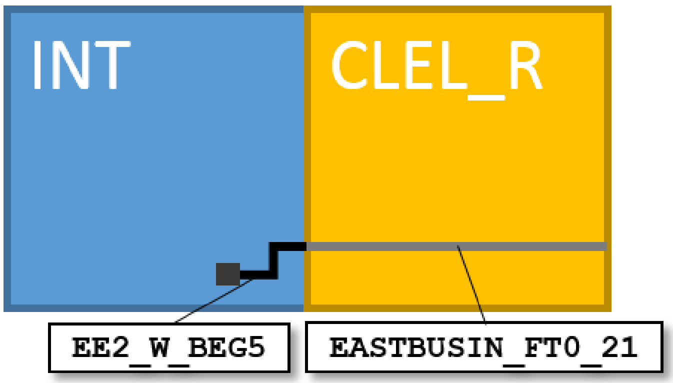


Fig. 5.3: An UltraScale+ SLICEL site, where logic BELs are magenta, routing BELs are green, site pins are red and site wires are yellow.

5.3 Tile

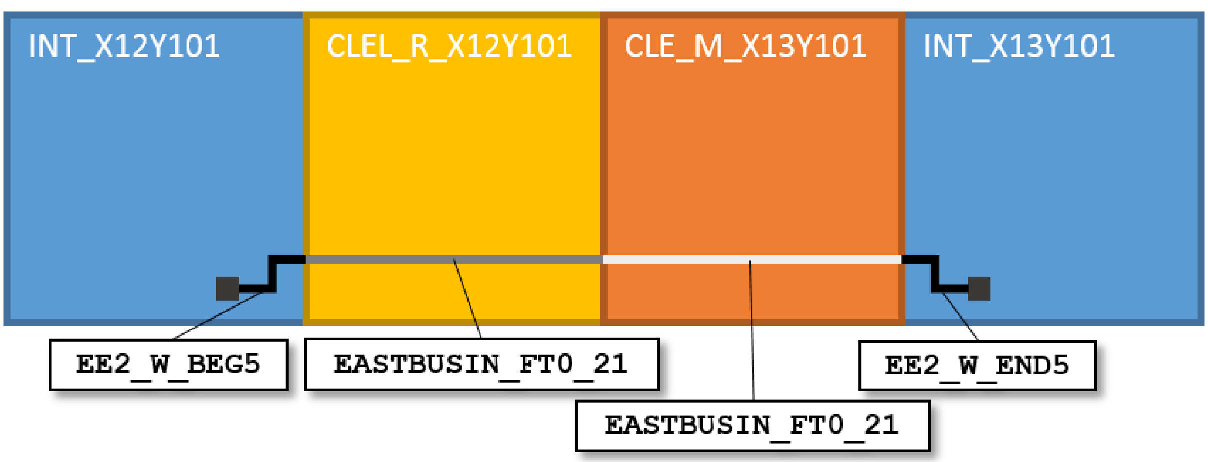
At an abstract level, Xilinx devices are created by assembling a grid of tiles. Similar to sites, each tile is an instance of a type and each tile has a unique name with an `_X#Y#` suffix. Tiles are the building blocks used when constructing an FPGA device. Tiles are designed to abut one another when laid down to construct an FPGA device.

Not all tiles contain sites and those that do, can have more than one. Unlike sites and BELs, tiles do not have user visible pins. Instead, tiles contain uniquely-named wires that can connect to site pins or other wires through a programmable interconnect point (PIP). PIPs are programmable muxes that connect two wires together in the same tile. Most PIPs are present in switch box tiles (those with the "INT" prefix). Columns of switch box tiles are designed to connect to all fabric resources such as CLBs, DSPs, and BRAMs. When tiles abut, they are designed such that certain wires in the adjoining tiles line up and connect as shown in the figure below:



5.3.1 Node

As there are no pins on tiles, the notion of a node is used to describe the connectivity of wires in between tiles. A node is a collection of electrically connected wires that spans one or more tiles. The figure below shows how four wires that abut among four tiles form a node:



Nodes and wires exist as first class Tcl objects in Vivado and the example above can be queried as follows:

```
% get_wires -of [get_node INT_X12Y101/EE2_W_BEG5]
INT_X12Y101/EE2_W_BEG5 INT_X13Y101/EE2_W_END5 CLEL_R_X12Y101/EASTBUSIN_FT0_21 CLE_M_
↪X13Y101/EASTBUSIN_FT0_21
%
```

For additional resources regarding Vivado objects, see UG912: Vivado Design Suite Properties Reference Guide.

5.3.2 Tile Type

Each tile belongs to a type or definition. A tile type will contain the inventory list of all wires, PIPs and site types. Vivado does not directly represent the tile type as an object, but is listed as a property value under each tile.

Xilinx traditionally has leveraged a columnar-based architectural approach to tile layout. That is, with a few exceptions, all tiles within a column are of the same type but tiles occupying the same row are typically different types.

5.4 FSR (Fabric Sub Region or Clock Region)

A fabric sub region, also known as a clock region, is a replicated 2D array of tiles in the fabric. In the UltraScale architecture, all FSRs are 60 CLBs tall, but their width will vary depending on the mix of tile types used in its construction.

Clock routing and distribution lines are represented as the same granularity as FSRs. In UltraScale architectures, there are 24 horizontal routing tracks, 24 vertical routing tracks, 24 horizontal distribution tracks and 24 vertical distribution tracks. These routing and distribution tracks abut to tracks in neighboring FSRs to form the device clock network resource set. For more information specific to clocking resources, please see UG472: Series 7 Clocking Resources User Guide or UG572: UltraScale Architecture Clocking Resources User Guide.

5.5 SLR (Super Logic Region)

This level of hierarchy is only present on devices that use the stacked silicon interconnect technology (SSIT) or also known as 2.5D packaging using a silicon interposer. As multiple dies (or dice) are packaged together, each die becomes a super logic region or SLR. SLRs contain a 2D array of FSRs and are typically identical as each die is fabricated from the same mask set.

In order for logic to communicate between SLRs, the UltraScale architecture employ special tiles in the FSRs neighbouring the abutment of two SLRs. A column of CLBs is removed and replaced with special tiles called Laguna tiles that have dedicated flip flop sites to aid in crossing the SLR divide.

5.6 Device

At the highest level of Xilinx architecture is the device. This is generally a 2D array of FSRs for single die products or two or more SLRs abutted vertically.

The core object in RapidWright is the Device class for any Xilinx device and is described in the next section.

CHAPTER SIX

RAPIDWRIGHT OVERVIEW

Table of Contents

This page aims to help bridge the gap between Xilinx architectural constructs and classes and APIs found within the RapidWright code base. There are three core packages within RapidWight: device, edif and design.

6.1 Device Package

The device package contains the classes that correspond to constructs in the hardware and/or silicon devices. The most prominent and important class in this package is aptly named the `Device` class. The `Device` class represents a specific product family member (xcku040, for example) but does not carry package, speed grade or temperature grade information. These additional unique attributes are captured in the `Package` class. When a specific device is combined with its package and grade information, this uniquely identifies a Xilinx part, represented by the `Part` class.

Most of the details of managing speed grades, packages, temperature are most commonly dealt with by using a string to uniquely identify a part is by using a String of the part name. RapidWright automatically interprets all valid and supported Xilinx devices by part name and can correctly load a device if that information is included or not. For example, the following lines of code all load the same device, even though the part name is slightly different:

```
Device device = null;
device = Device.getDevice("xcku040");
device = Device.getDevice("xcku040-fbva676-2");
device = Device.getDevice("xcku040ffva1156");
device = Device.getDevice("xcku040-sfva784-1LV-i");
device = Device.getDevice("xcku040ffva1156-2");
```

The `Device` class maintains a singleton map to avoid loading the same device more than once. Devices files are stored in `com.xilinx.rapidwright.util.FileTools.DEVICE_FOLDER_NAME` and are provided by the maintainers of the RapidWright project, typically refreshed with each production release of Vivado (2017.3, 2017.4, 2018.1, . . .). A significant amount of information is stored in the device files and so they are highly compressed to avoid consuming excessive disk space.

The Device class makes available all of the architectural resources through various APIs and data objects that follow the same hierarchical model as shown previously in the *Xilinx Architecture Terminology* section. For convenience, here again is the logical hierarchy of Xilinx devices:

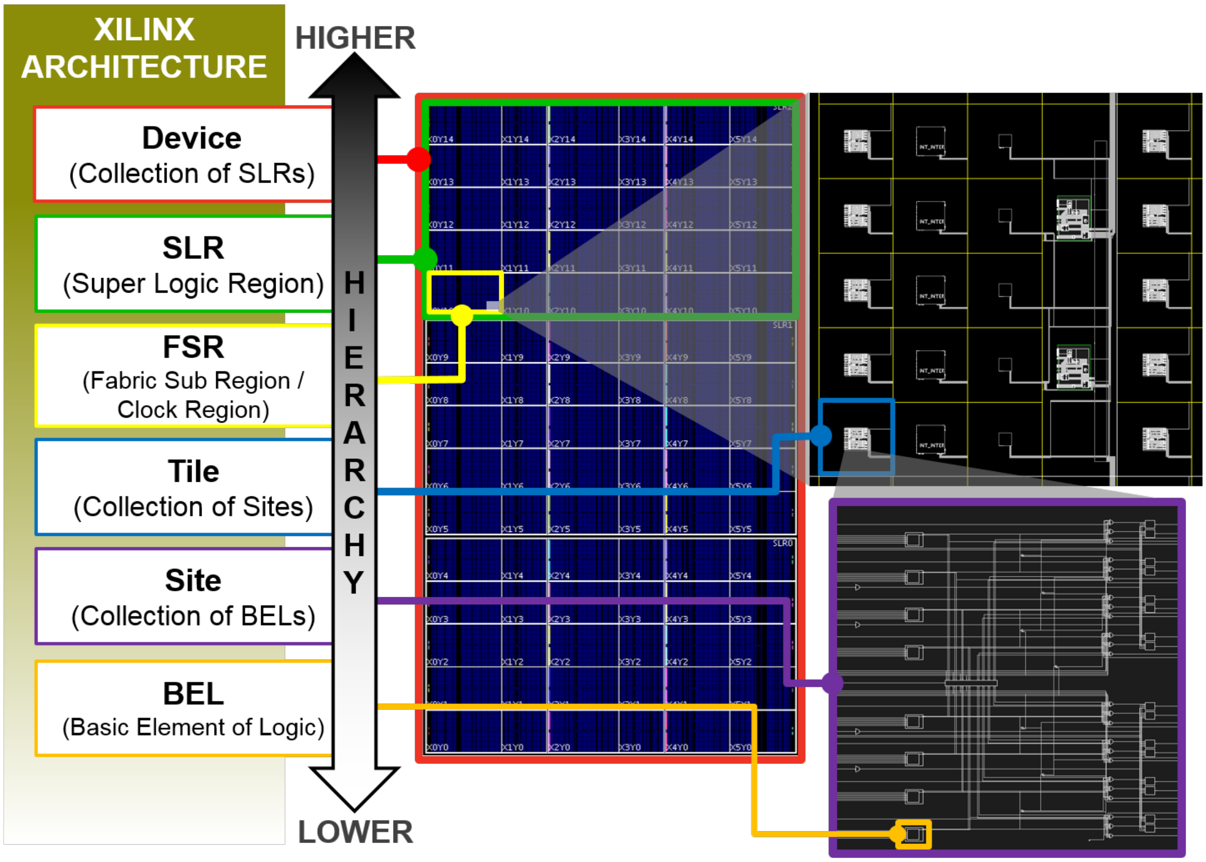


Fig. 6.1: Levels of architectural hierarchy in Xilinx FPGAs.

These levels of hierarchy are available in RapidWright and the table below shows basic getters in both RapidWright and Vivado.

RapidWright Class	RapidWright Java API	Vivado Object	Vivado Tcl API
SLR	Device.getSLR(int id)	SLR	get_slrs -filter SLR_INDEX==$idx
ClockRegion	Device.getClockRegion(int row,int col)	Clock Region	get_clock_regions -filter NAME==$name
Tile	Device.getTile(String name)	Tile	get_tiles -filter NAME==$name
Site	Device.getSite(String name)	Site	get_sites -filter NAME==$name
BEL	Site.getBEL(String name)	BEL	get_bels -of $site -filter NAME==$name

The `Device` class is the top level object in RapidWright and has direct accessors to all other levels of hierarchy except for BELs. All classes in the hierarchy are static and do not change based on a user design. Most of the interaction between a user's design and the device occur at the Tile, Site and BEL levels of hierarchy. The `BEL` class can be one of three kinds of non-routing objects in a `Site`: a Logic BEL, a Routing BEL and a Port (port of the Site). This is designated by its class member enum of type `BELClass`. Most components within the device architecture are assigned an integer index. This helps to lower memory usage by not always having to explicitly represent a component of the architecture with a dedicated object. It also helps by providing faster lookups. In some cases, such as `TileTypeEnum` and `SiteTypeEnum`, the index has been explicitly enumerated and an enum is used instead.

In parallel with the logical hierarchy of Xilinx devices, there also exist several constructs for representing routing resources. At the lowest level are pins on BELs represented by the `BELPin` class. Pins on `Site` objects can be referenced by creating dynamic objects of type `SitePin`. Inside a `Site`, wires called 'site wires' connect `BELPin` objects. Connectivity of a site wire is stored in each `BELPin` and also in the `Site` object. Site wires do not have an explicit object for representation, but their name, index and connectivity are available on `Site` and `BELPin` objects.

Remaining faithful to the Vivado representation of inter-site routing resources, RapidWright provides `Wire`, `Node` and `PIP` (Programmable Interconnect Points) objects. These objects are generated on the fly as needed as there can be several millions of unique instances of each. The figure below correlates a Vivado device GUI representation with an example of the different routing resources types available in RapidWright.

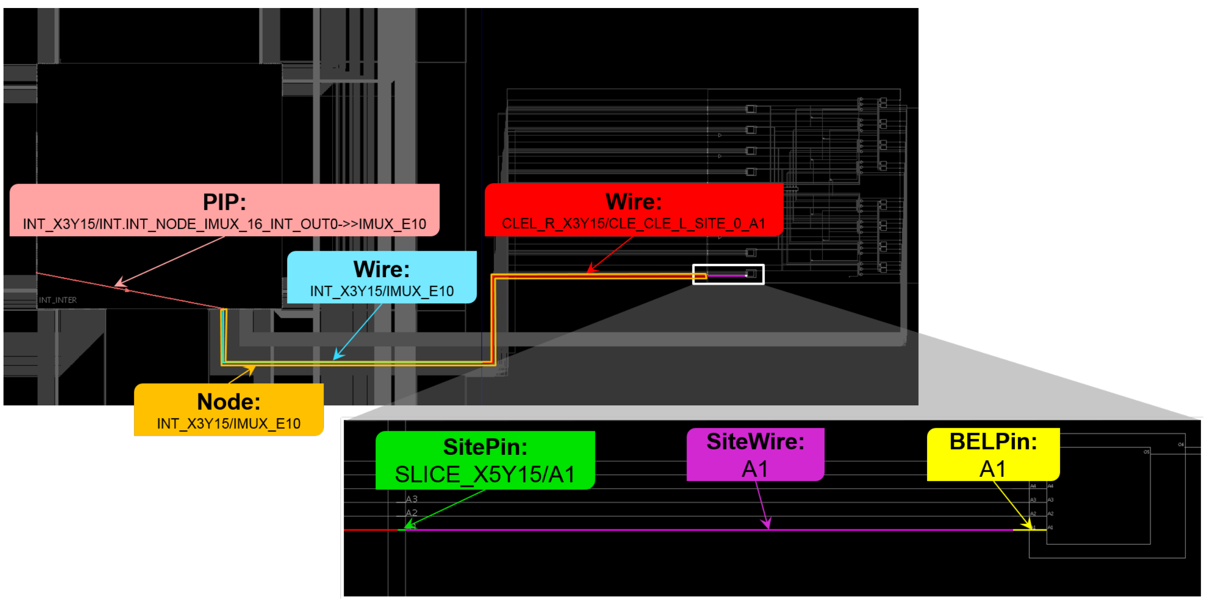


Fig. 6.2: Examples of different routing resources Xilinx FPGAs.

6.2 EDIF Package (Logical Netlist)

In Vivado, all designs post synthesis have a logical netlist that can be exported in the EDIF netlist format. EDIF (Electronic Design Interchange Format) 2 0 0 is the netlist format used in RapidWright. This is due to its inclusion in Vivado's design checkpoint file format and that Vivado has facilities to read and write it (`read_edif` and `write_edif`).

RapidWright reads, represents and writes logical netlist information in the EDIF format and the EDIF package is written to explicitly accommodate this need. It was written with Vivado-generated EDIF in mind and may not support every corner case of the EDIF 2 0 0 specification.

Parsing EDIF is performed by the `EDIFParser` class. EDIF is normally handled when reading or writing a DCP, but it can be parsed/exported independently as follows:

```
// Read in my_edif_file.edf
EDIFParser parser = new EDIFParser("my_edif_file.edf");
EDIFNetlist netlist = p.parseEDIFNetlist();
// Work some netlist magic...
// ...
```

```
// Now write it out
netlist.exportEDIF("my_edif_file_post_rapidwright.edf");
```

The `EDIFNetlist` is the top level class that contains the netlist and cell libraries. All EDIF-related objects have EDIF has a class name prefix. The `EDIFNetlist` keeps a reference to the top cell which is wrapped in the `EDIFDesign` class. It also maintains a top cell instance reference that is generated when the file is loaded.

Although a full explanation of netlist modeling and relationships are beyond the scope of this documentation, an attempt to clarify the contextual meaning of some of the classes will be made. One important distinction to make is between `EDIFPort` and `EDIFPortInst`. At one level, an `EDIFPort` belongs to an `EDIFCell` and an `EDIFPortInst` belongs to an `EDIFCellInst`. Another distinction is that an `EDIFPort` can be a bussed-based object whereas an `EDIFPortInst` can only represent a single bit. An `EDIFNet` defines connectivity inside an `EDIFCell` by connecting `EDIFPortInst` objects together (port references on cell instances inside the cell or to external port references entering/leaving the cell).

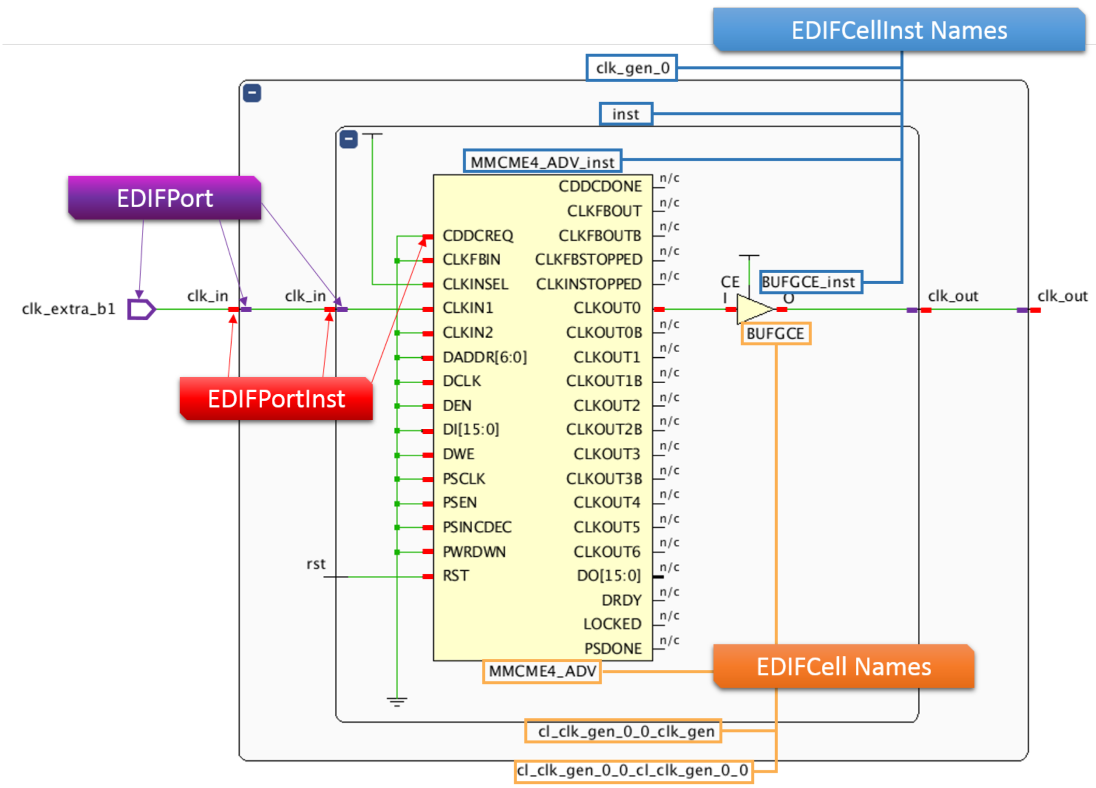


Fig. 6.3: Snapshot of the Vivado netlist viewer with references to RapidWright EDIF classes

Most classes inherit from `EDIFName`. EDIF has peculiar naming rules and provides for a mechanism to map the original name to a legal EDIF name. The EDIF package in RapidWright attempts to hide all of the String gymnastics necessary to maintain both name spaces and simply present the user with the original intended name.

Several classes also inherit from `EDIFPropertyObject` (which also inherits from `EDIFName`). `EDIFPropertyObject` endows objects with the ability to store properties which are key/value pairs. Properties are a mapping between an `EDIFName` object and a `EDIFPropertyObject`. These properties can contain key programmable information such as LUT equations or attributes specific to BEL sites.

6.3 Design Package (Physical Netlist)

The design package is the collection objects used to describe how a logical netlist map to the device netlist. The design is also referred to as the physical netlist or implementation. It contains all of the primitive logical cell mappings to hardware, specifically the cells to BEL placements and physical net mapping to programmable interconnect or routing.

The `Design` class in RapidWright is the central hub of information for a design. It keeps track of the logical netlist, physical netlist, constraints, the device and part references among other things. The `Design` class is most similar to a design checkpoint in that it contains all the information necessary to create a DCP file.

Since a design programs a device, there are some one-to-one mappings between the device and design representation in RapidWright. For example:

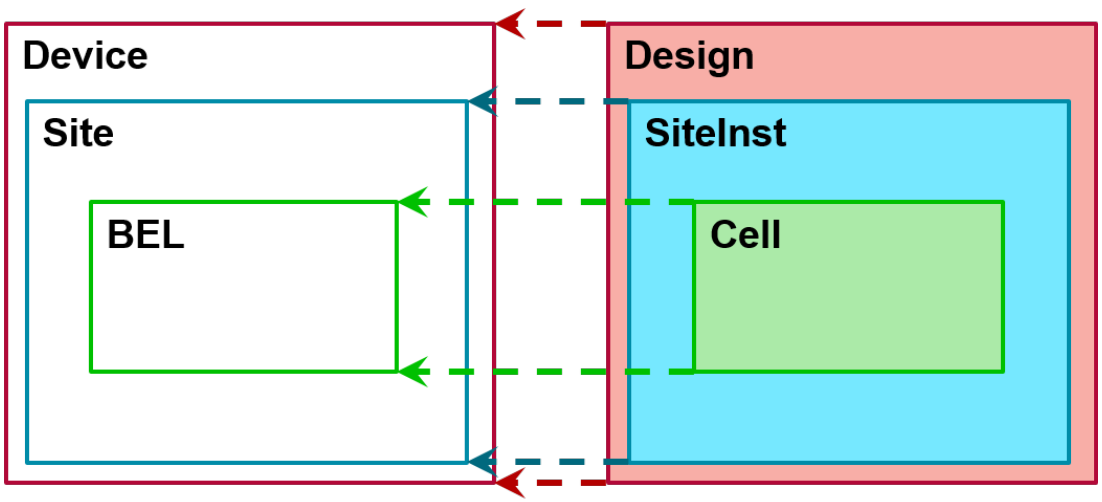


Fig. 6.4: Illustration representing how a `Cell`, `SiteInst` and `Design` map to `BEL`, `Site` and `Device` respectively

6.3.1 SiteInst

Design representation and implementation in Vivado is BEL-centric (BELs and cells). The `SiteInst` keeps track of the cells placed onto its BELs, the site PIPs used in routing and how routing resources map to nets.

Each `SiteInst` maps to a specific compatible site within a device. The `SiteInst` has a type using a `SiteTypeEnum` as the designator. It also maintains a map of named leaf cells from the logical netlist that are physically placed onto the BEL sites within the site. RapidWright also preserves the same Vivado "fixed" flag that is used in certain situations by Vivado to prevent components insides the site from being moved.

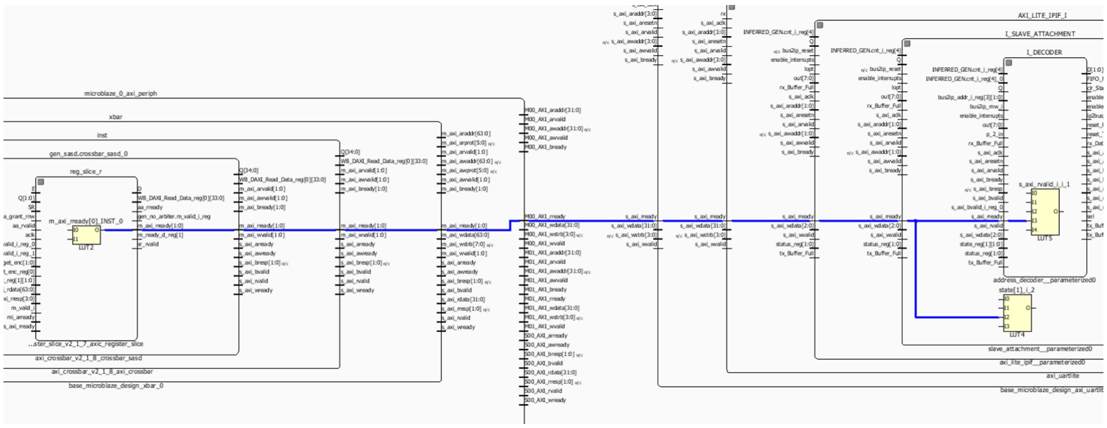
Routing nets inside of a site (intra-site) is different from routing outside of sites (inter-site). Routing nets outside of sites consists of finding a path of `Node` objects from a source site pin to a sink site pin by turning on a set of PIPs. In contrast, routing inside of a site can be a bit more complex as it must also account for site context and consider which BELs are occupied. In general, Vivado attempts to automate the intra-site routing task. RapidWright also strives to do the same (see SiteInst.routeSite()), however it may not always fully automate tasks as expected and the user may be required to call additional APIs when placing/routing design elements.

One of the ways routing is accomplished inside a site is through a `SitePIP`, which is a programmable interconnect point that exists on a routing BEL. Generally, a `SitePIP` will establish a connection through a routing BEL or, in some cases, a logic BEL from an element input pin to an element output pin, thus connecting two separate site wires.

The `SiteInst` is the object in RapidWright where site PIP usage is recorded and maintained. By default all site PIPs are turned off, if the site PIP is added to the `SiteInst` then it is interpreted as the site PIP being turned on or used.

6.3.2 Net

Routing outside of a site is represented by the `Net` class. A `Net` in RapidWright is typically named after the logical driver source pin and represents the entire set of logically equivalent nets that map to the same electrically equivalent net. For example, consider the net depicted in the following netlist screenshot:



This figure shows the logical netlist connection of three cells over one physical net. However, there are 11 separate nets in the logical netlist that must be traversed in order to make the connection.

A `Net` is a physical net that implements a route using PIPs (programmable interconnect points) that, when combined together connect nodes into a path from a source site pin to one or more sink site pins. A `Net` starts and stops at site pins represented by `SitePinInst` objects (design instances of `SitePin` objects). The physical implementation of the 11 logical nets above is shown in the figure below:

The net is also referenced when routing inside a site, but the site routing implementation is captured in the `SiteInst` object.

6.3.3 Cell (A BEL Instance)

At the lowest level, a RapidWright `Cell` maps a logical leaf cell from the EDIF netlist (`EDIFCellInst`) to a `BEL`. The cell name is typically the full hierarchical logical name of the leaf cell it maps to and also maintains the library cell type name (FDRE, for example for a reset flip flop). A cell also maintains the logical cell pin mappings to the physical cell pin mappings (pins on the `BEL`).

6.3.4 Module

A module is a physical netlist container construct available in RapidWright. A RapidWright module is represented by the `Module` class in the `design` package. A module contains both a logical and physical netlist that provides all the details necessary for a full implementation. It is most similar to a placed and routed out-of-context DCP, however RapidWright enables the implementation to be replicated or relocated to multiple compatible areas of the fabric—capabilities that are not yet available in Vivado. A module is a definition object in that the `SiteInst` and `Net` objects it contains are a prototype or blueprint for a pre-implemented block that can potentially be 'stamped' out and relocated in valid locations around a device. The `ModuleInst` represents the instance object of a `Module` and is part of the implemented portion of a physical netlist.

6.3.5 Module Instance

A module instance quite simply is an instance of a module. RapidWright supports module instances in a design using the `ModuleInst` class in the `design` package. Module instances have a unique name within the design and as each module has a collection of `SiteInst` and `Net` objects, these containers are prefixed hierarchically with the module instance name. For example, if a module had a `SiteInst` named "SLICE_X2Y2" and a `Net` named `data_ready`, a newly created module instance named "fred" would have counterpart `SiteInst` and `Net` objects called "fred/SLICE_X2Y2" and "fred/data_ready".

A module instance will typically have one of its site instances selected as what is called an 'anchor'. The anchor site instance is a common reference point by which all other site instances and nets in the instance can be referenced. This is useful for determining if a potential location on the fabric is compatible with the module instance for placement.

The `Module` and `ModuleInst` concept is not available in Vivado or the DCP file format. If these constructs are used in a RapidWright design they will be 'flattened' when written out as a DCP.

CHAPTER

SEVEN

DESIGN CHECKPOINTS

Table of Contents

7.1 What is a Design Checkpoint?

A design checkpoint (DCP) is a file used by the Vivado Design Suite that represents a snapshot of a design at any stage of the design process. The snapshot includes the netlist, constraints and implementation results.

7.2 What is Inside a Design Checkpoint?

A design checkpoint file (extention .dcp) is a Vivado file format that contains a synthesized netlist, design constraints and can contain placement and routing information. RapidWright provides readers and writers to parse and export the various components.

7.3 RapidWright and Design Checkpoint Files

RapidWright can freely read and write checkpoint files with the following exceptions:

- If the design is encrypted, RapidWright cannot open it. RapidWright is not capable of decrypting files.
  - Sometimes, however, a design may not be secured or designated to be encrypted but the EDIF file in the DCP is encrypted. This is due to RTL source references being stored in the EDIF file. Vivado will allow you to write out an EDIF file (without RTL source references) with the `write_edif` Tcl command. RapidWright can read in the alternate EDIF file along side the DCP if it has the same root name (.edf extension instead of .dcp).
- If the design checkpoint file is created with a much newer version of Vivado compared with the RapidWright release, it may not be able to read the file.
- Conversely, older versions of Vivado may not be able to read RapidWright checkpoint files

Here are a few ways to read/write a design checkpoint in RapidWright:

```
Design design = Design.readCheckpoint("my_design_routed.dcp");
// or if the EDIF inside the DCP is encrypted because of source references,
//   you can alternatively supply a separate EDIF
design = Design.readCheckpoint("my_design_routed.dcp", "my_design_edif.edf");

// To write out a design
design.writeCheckpoint("my_design_post_rapidwright.dcp");
```

The interface that enables RapidWright to read and write checkpoints is handled by the RapidWright API Library in the provided rapidwright-api-library-<ver>.jar. The APIs in this tool are used in the Design class with readCheckpoint() and writeCheckpoint(). Note that it is licensed separately from the rest of RapidWright under a modified Xilinx EULA. Also note that RapidWright is not an official product from Xilinx and designs created or derived from it are not warranted. Please see LICENSE.TXT for full details.

CHAPTER

EIGHT

IMPLEMENTATION BASICS

Table of Contents

Implementation, in the context of RapidWright and compiling designs for FPGAs, is defined as the placement and routing of a synthesized/mapped netlist to a specific FPGA device. This section will describe the detailed mechanics of how placement and routing can be achieved in RapidWright.

8.1 Placement

As opposed to Vivado, RapidWright enables three layers or levels of placement in its design abstraction: BEL level, site level and module level. Vivado primarily only enables BEL placement (previously in ISE, sites were the major unit of placement). This section details how RapidWright represents and interacts with design elements at the three levels of placement mentioned.

8.1.1 BEL Placement

Note: Reliable automatic BEL placement in RapidWright is still a work in progress and care should be taken when attempting this capability.

Creating correct BEL placements is quite tricky as several factors must be taken into consideration when placing a cell onto a BEL site. Some questions one might need to ask when placing a cell onto a BEL site are:

1. Is the BEL site already occupied and are all pins map-able to the surrounding BEL connections?
2. Are all of the cell connections routable within the site and interconnect?
3. Are the clock and set/reset domains compatible with those already used within the site or are there resources available to route alternatives?
4. Does this cell depend on any dedicated inter-site wires (such as carry chains or DSP cascades) that are not available?

Placing a cell correctly can necessitate updates to the design in the following categories:

1. Mapping of a `Cell` object to a `BEL` in RapidWright
2. Pin mappings between the logical and physical cell pins must be added and/or routed within the site (conditions will vary).
3. Use of one or more SitePIPs as part of routing the site (stored in the respective `SiteInst`)

Generic pin mappings are assigned when a cell is created and placed. However, these mappings may need to be adjusted based on the context.

A SitePIP configures a routing BEL to propagate a signal from one of its inputs to its output pin. SitePIPs must be turned on in the respective `SiteInst` when a cell is placed onto a BEL as the common convention in Vivado is to always leave the site in a legally routed state.

8.1.2 Site Placement

Within RapidWright, it can be straightforward to move a `SiteInst` from one site to another. An example of how to relocate a site instance from one location to another is shown below:

```
Design d = Design.readCheckpoint("example.dcp");
SiteInstance si = d.getSiteInstanceFromSiteName("SLICE_X0Y0");
si.place(d.getDevice().getSite("SLICE_X1Y1"));
```

The user is responsible for changing any existing routing resources that previously routed to the old site.

8.1.3 Module Placement

One of RapidWright's unique capabilities is providing another level of hierarchy in implementation. Through the `Module` and `ModuleInstance` classes, a complex cell can be replicated and/or relocated across the device. When a pre-implemented module is created for a device, all valid locations are pre-calculated and stored for the anchor site within the `Module`. Therefore, placement of a `ModuleInstance` is simply selecting one of the valid anchor sites and applying it.

8.2 Routing

In Vivado, there is roughly three different types of routing: intra-site, inter-site and clock routing. This section provides a brief overview of each.

8.2.1 Site (Intra-site) Routing

When a cell is placed onto a BEL, typical Vivado convention is to route the intra-site net portions immediately after. Routing a site implies mapping the physical net to site wires and site PIPs. In RapidWright, some of this intra-site routing happens when the cell is placed and there are a few methods that can also help finish intra-site routing in special cases. `SiteInst.routeIntraSiteNet()` will attempt to route one BELPin to another for intra-site nets. `SiteInst.routeSite()` will attempt to route all the nets that pertain to the site.

8.2.2 Interconnect (Inter-site) Routing

The majority of work in routing a design is in inter-site routing. This is the task of selecting a set of routing resources the enable a path between a source site pin and one or more sink site pins. The physical routing of a net in RapidWright is simply described by a list of PIPs. RapidWright comes with a rudimentary router for UltraScale architectures, but it is still a work in progress. It doesn't fully resolve congestion, but provides a working example for more specialized tasks.

8.2.3 Clock Routing

Clock routing is very architecture specific and is similar to inter-site routing in that it is also implemented by a list of PIPs. However, there are key steps and constraints that must be satisfied beyond typical inter-site routing.

(More to come…)

CHAPTER

NINE

A PRE-IMPLEMENTED MODULE FLOW

This section describes a pre-implemented module flow that can operate in two ways:

1. Target high performance implementations by reusing high quality, customized solutions.
2. A rapid prototyping demonstration vehicle that hints at a future of fast compile times.

9.1 Background and Flow Comparison

Both flows (high performance and rapid prototyping) start with the RapidWright provided Tcl command, `rapid_compile_ipi`. This command can be loaded by running `source ${::env(RAPIDWRIGHT_PATH)}/tcl/rapidwright.tcl` in the Vivado Tcl interpreter. Optionally, you can also configure Vivado to source the script each time it starts by modifying the Vivado_init.tcl (see the section 'Loading and Running Tcl Scripts' in UG894: Vivado Design Suite User Guide - Using Tcl Scripting).

Note: If you are using a standalone jar, you can extract the rapidwright.tcl (and other device/data) by running `java -jar <standalone.jar> --unpack_data` and setting the environment variable RAPIDWRIGHT_PATH to the standalone jar location.

This command runs on an open IP Integrator design by synthesizing, placing and routing all IP blocks out-of-context (OOC). Each block is provided a pblock (area constraint before placement to improves its re-usability). The implemented result for each IP is stored in the Vivado IP cache. RapidWright then uses the cache for each subsequent run (and only pre-implements one of each kind of IP—so if your design has multiple instances, only one run per type). After all IPs have been implemented OOC, it invokes the `BlockStitcher` in RapidWright to stitch all of the pre-implemented blocks together, places the blocks and routes them into a final implementation (note: currently RapidWright router is disabled). This command, can function in two modes as described previously. Here is a quick comparison of the high performance vs. rapid prototyping mode for pre-implemented blocks:

	High Performance Flow	Rapid Prototyping Flow
PBlock Selection	Application Architect (Manual)	PBlock Generator
Block Placement	Application Architect (Manual)	Block Placer
Global Routing	Vivado	RapidWright Router OR Vivado

The high performance flow (as described in more detail in the *High Performance Flow* section below) requires input from the application architect of the design. This does involve extra effort, but leads to potentially the highest implementation results. The *Rapid Prototyping Flow* is optimized more for fast compile times by automating the tasks of pblock selection for each block/IP involved and also placement of the blocks.

9.1.1 Module Cache

In order to better facilitate fast loading performance of modules, RapidWright has a fast and efficient file format for storing modules in a directory called a cache. The facilities for reading and writing these module storage files are found in the `BlockCreator` class found in the `ipi` package. As each IP to be implemented in a design might have different physical contexts or placement pblocks, multiple implementations of the same `Module` are stored in a `ModuleImpls` object which is simply an extended `ArrayList<Module>`. This allows all the implementations to reside in the same object and file and to reference each unique implementation with an index. Each RapidWright module entry has three relevant files:

1. Input: A metadata text file generated from Vivado to communicate information about the IP, its ports, clocks, constraints and approximate delays on inputs and outputs. This file is read into RapidWright during the module file creation process.
2. Output: To store the physical implementation data of each module implementation, a '.dat' file is created from `BlockCreator`.
3. Output: The logical netlist is shared among all implementations and is stored in a compressed EDIF file format with a '.kryo' extension.

The RapidWright module cache builds on top of the IP cache in Vivado. By default RapidWright puts the cache in the $HOME/blockCache directory. This can be changed by setting the environment variable `IP_CACHE_PATH` before running the flow.

The IP cache generated by Vivado is supplemented by RapidWright by providing placed and routed DCPs and module files in each hash-named directory for each non-trivial IP. By default, the flow only creates a single implementation for each IP. Later, we describe how a user can create an implementation guide file ('.igf') directing the flow to create multiple unique implementations of the same module/IP.

9.1.2 Block Stitcher

The block stitcher (found in the class `BlockStitcher` of the `ipi` package) is the heart of the pre-implemented design flow. It manages the flow progress and ensures that all blocks have been cached and retrieved appropriately. It also reads in the IP Integrator netlist file (EDIF) that describes the block connectivity and stitches together the block implementations in the physical netlist. It also reads and parses the implementation guide file (if provided) and creates the block implementations accordingly.

9.2 High Performance Flow

One of the key attributes of RapidWright is the ability to capture optimized placement and routing solutions for a module and reuse them in multiple contexts or locations on a device. Vivado often provides good results for small implementation problems (smaller than 10k LUTs within a clock region). However, when those same modules are combined into a large system, total compile time increases and the probability of timing closure is reduced. This phenomenon limits achievable performance and timing closure predictability of larger designs.

RapidWright endows users with a new design vocabulary by caching, reusing and relocating pre-implemented blocks. We believe this to be an enabling concept and offer a three-step high performance design strategy:

1. Restructure the Design: Expose all modular pieces and replication in an IP Integrator design.
2. Packing & Placement Planning: Craft custom pblocks and placement patterns to match architecture layout and resources.
3. Stitch, Place & Route Implementation: Run the automated flow to create a final implementation.

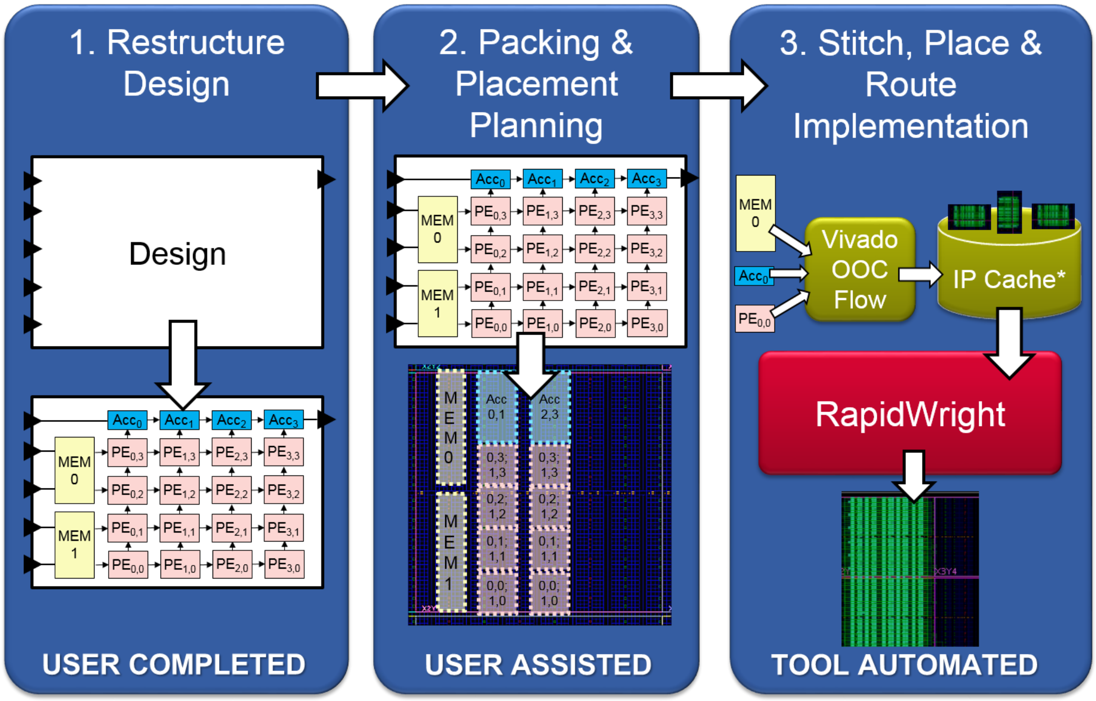


Fig. 9.1: High level visual of the three step process for the high performance module-based design strategy

The first step requires the design architect to restructure the proposed design such that it can take full advantage of the benefits provided by pre-implemented modules. We define restructuring as a design refactoring that reflects three favorable design characteristics: (1) modularity, (2) module replication and (3) latency tolerance. Modularity uncovers design structure so it can be strategically mapped to architectural patterns. When modules are replicated, reuse of those high quality solutions and architectural patterns can be exploited to increase the benefits. Finally, if the modules within a design tolerate additional latency, inserting pipeline elements between them improves both timing performance and relocatability.

After the design architect has successfully restructured and modularized a design, step two is followed. Here, the design architect creates an implementation guide file that captures how best to map the modules of a design to the architecture of the target device. Specifically, pblocks are chosen for those pre-implemented modules of interest and physical locations are chosen for each instance. This step provides the design architect an opportunity to navigate FPGA fabric discontinuities. These discontinuities include boundaries such as IO columns, processor subsystems, and most significantly, SLR crossings. Such architectural obstacles cause design disruptions when targeting high performance. However, by leveraging a pre-implemented methodology provided in RapidWright, custom-created implementation solutions can be identified and planned out to manage the fabric discontinuities by custom module placement. Ultimately, this process is iterative and can inform useful RTL/design changes by focusing design structure to better match architectural resources.

Step three of the design strategy is an automated flow provided with RapidWright (depicted in the diagram above). We leverage a design input method in Vivado called IP Integrator (IPI). IPI offers an interactive block-based approach for system design by providing an IP library, IP creation flow and IP caching. RapidWright takes advantage of IPI by using leaf IP blocks as de-facto pre-implemented blocks and also by leveraging the IP caching mechanism. The RapidWright pre-implemented flow extends the caching mechanism to go beyond synthesis, by performing OOC placement and routing on the block within a constrained area. The flow begins by invoking Vivado's typical IPI synthesis and creating pre-implemented blocks for each module if not already found in the cache. RapidWright has an IPI Design Parser (EDIF-based) that creates a black-box netlist where each instance of a module is empty, ready to receive the pre-implemented module guts. The block stitcher reads the IP cache and populates the IPI design netlist. After stitching, the blocks are placed either by loading the implementation guide file or invoking a simulated annealing

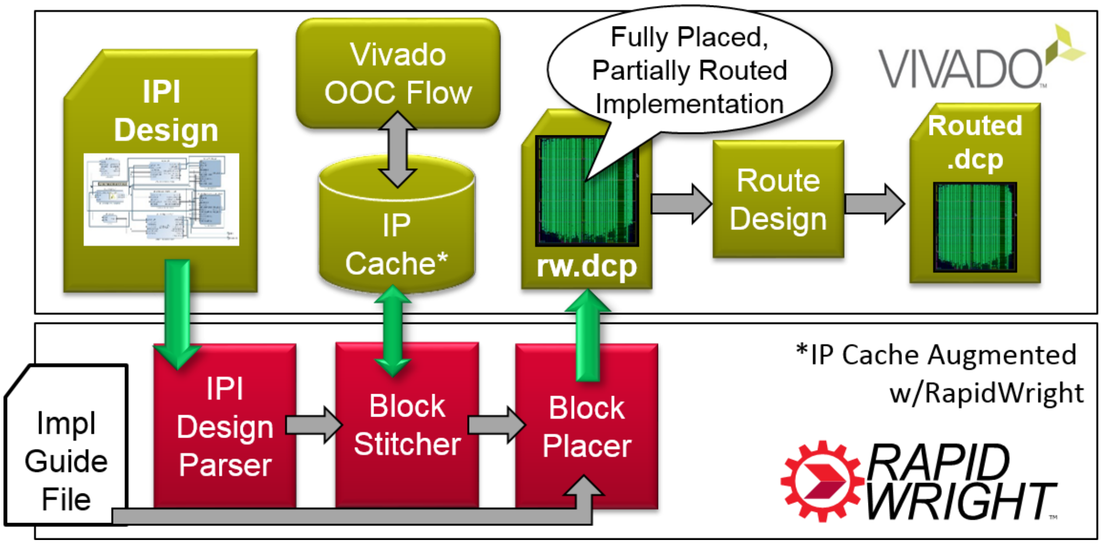


Fig. 9.2: High level view of the pre-implemented flow process and interactions between Vivado and RapidWright

module placer to place the blocks onto the fabric automatically. Once all the blocks are placed, RapidWright creates a DCP file that is read into Vivado which completes the final routes.

9.2.1 Implementation Guide File

An implementation guide file (extension *.igf) allows the application architect to communicate all of the specific implementation customization aspects of the packing and placement phase. The file has the following syntax structure (note the use of … which indicates a potential repetition of the previous construct):

```
PART <part_name>
BLOCK <ip_cache_id> <# of implementations> <# of instances in the design> <# of␣
↪clocks used in this block>
IMPL <implementation index> [# of sub implementation entries] <Pblock range>
        [SUB_IMPL <sub implementation index> '<Tcl command returning a subset of␣
↪cells in the module>' <pblock range>]
        ...
...
INST <instance name> <implementation index to apply> <lower left corner site to place␣
↪implementation on fabric>
...
CLOCK <clock name> <clock period constraint (ns)> <BUFGCE site (to use for skew␣
↪estimation)>
...
END_BLOCK
...
END_BLOCKS
```

A parser and export for the IGF format can be found in `com.xilinx.rapidwright.design.blocks.ImplGuide.readImplGuide(String fileName)` and `com.xilinx.rapidwright.design.blocks.ImplGuide.writeImplGuide(String fileName)`.

BLOCK (IP Cache Entry)

The block construct describes all of the potential implementations for a particular block/IP. For each uniquely configured IP (entry in the IP cache), there exists a block. Multiple instances of the same block/IP can exist and this construct allows the application architect to map instances by name to a specific implementation.

IMPL (Implementation)

Each block has one or more IMPLs. Each implementation carries a pblock and potentially some SUB_IMPL which allows for sub pblocks to be applied to portions of the logic inside the block. Each IMPL is indexed so that it can be referenced and applied to specific instances of the block. The application architect takes special care in selecting implementations and their pblocks to maximize there potential performance, architectural footprint and placement packing efficiency.

SUB_IMPL (Sub Implementation)

This is an optional construct that allows for more fine-grained pblocks being applied to a partial subset of the block/IP in an implementation. One field requires a Tcl command that returns a subset of cells that should be included in the sub implementation and associated pblock. Multiple sub implementation entries can exist for each implementation. As an example, if a particular IP is tall and narrow and there are specfic cells that need to be placed at the top and/or bottom, the SUB_IMPL contruct can be used to pblock the top and bottom specific cells in sub pblock of the overall implementation.

INST (Instance)

In each design, there will be one or more instances of a block/IP. Each instance has a unique name and must be assigned to an implementation. Each instance also requires a placement which is provided by denoting a specific site onto which the lower left corner of the pblock of the respective implementation could be placed.

CLOCK (Clock Input)

The clock construct describes a clock input to the block or IP and allows it to apply a clock period constraint in nanoseconds. It also requires the BUFGCE site from which the clock will be driven so that during placement and routing, the clock skew can be estimated.

Basic Example

The diagram below illustrates a basic BLOCK example with many of the different fields highlighted.

9.3 Rapid Prototyping Flow

When an implementation guide file is not provided when calling the `rapid_compile_ipi` command, the flow defaults into a rapid prototyping flow that targets faster compilation. As no user input is provided to guide pblock selection or block placement, RapidWright provides automated facilities that accomplish these tasks automatically, albeit with lower average performance than the application architect.

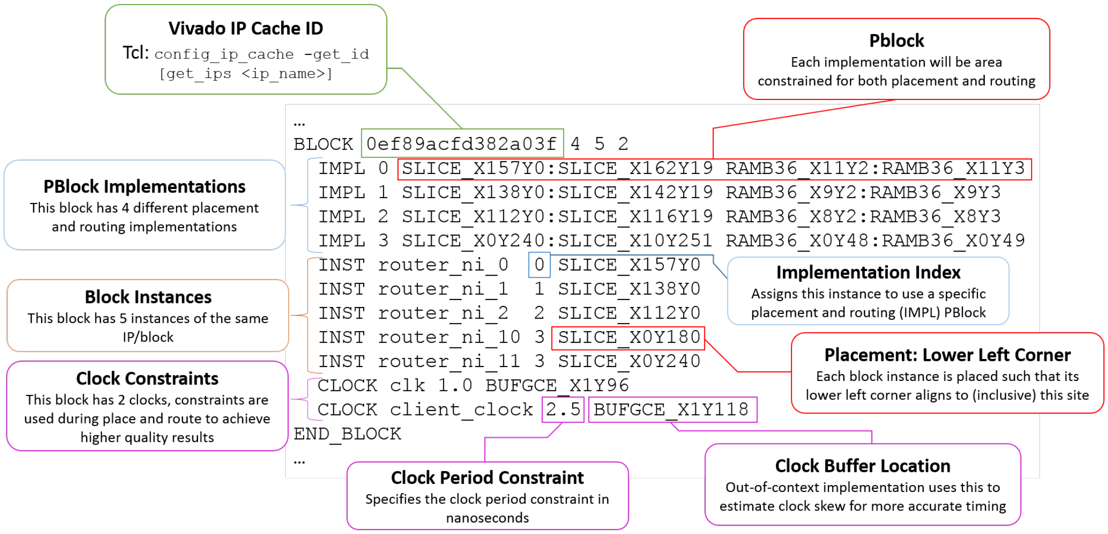


Fig. 9.3: BLOCK example with multiple implementations, instances and clocks

9.3.1 Automatic PBlock Generator

The automatic pblock generator is found in the `design.blocks` package in the class called `PBlockGenerator`. It takes as input two files to calculate an appropriate pblock for a given circuit. First it uses a utilization report file (produced by Vivado's `report_utilization` command) to identify the types of resources needed and their quantity. Second, it reads a shapes report file that describes all of the shapes in the design to ensure that the pblock size can easily accommodate all shapes. Shapes are an internal Vivado construct to help small groups of cells be placed together (such as carry chains). In the pre-implemented flow, the PBlockGenerator is always invoked for each IP that is created, specific Tcl commands are found in the `tclScripts/rapidwright.tcl` file in the `compile_block_dcp` proc.

One of the techniques used by the PBlockGenerator is to identify the most common tile column patterns (see `TileColumnPattern` class in the `device.helper` package) found in a particular device and place the pblock onto the most common match for a given resource footprint to maximize the place-ability of the block.

Expectations for performance should be muted as the prioritization for the pblock generator is to produce a pblock that won't cause place and route to fail and lacks knowledge of the particular context of the design where the block may be destined. For this purpose, it is highly recommended that any performance critical block or design use the implementation guide file as a way to better optimize the pblock for a particular application.

9.3.2 Block Placer

The Block Placer (found in the class `BlockPlacer2` of the package `placer.blockplacer`), uses a simple simulated annealing schedule to place the blocks on to the fabric. The cost function is a function of total wire length between blocks. Again, like the pblock generator, the block placer attempts to produce valid results, with less emphasis on performance.

9.3.3 Router

The router is a very simple maze router with very limited routing congestion avoidance. Its clock router is still a work in progress and is currently disabled. It is currently tuned to work with UltraScale and UltraScale+ architectures. The `Router` class is found in the `router` package.

RAPIDWRIGHT TUTORIALS

10.1 Create Placed and Routed DCP to Cross SLR

What You'll Need to Get Started:

- RapidWright 2018.2 or later
- Vivado 2018.2 or later

One of the example programs that is provided with RapidWright solves a challenging problem on UltraScale+ devices (this approach is not valid for Series 7 or UltraScale parts). Crossing super logic region (SLR) boundaries at high speed can prove quite difficult in conventional Vivado flows. The hardware provides dedicated TX/RX flip flops in Laguna sites to enable the creation of paths with very short delay but experience two significant problems:

1. The dedicated super long lines (SLLs) that connect TX and RX Laguna flip flop pairs are often sensitive to hold time violations due to the higher multi-die variability.
2. Paths crossing the SLR boundary are taxed with an additional delay penality called "Inter-SLR Compensation" (ISC). This penalty increases the calculated delay and reduces it potential for high speed.

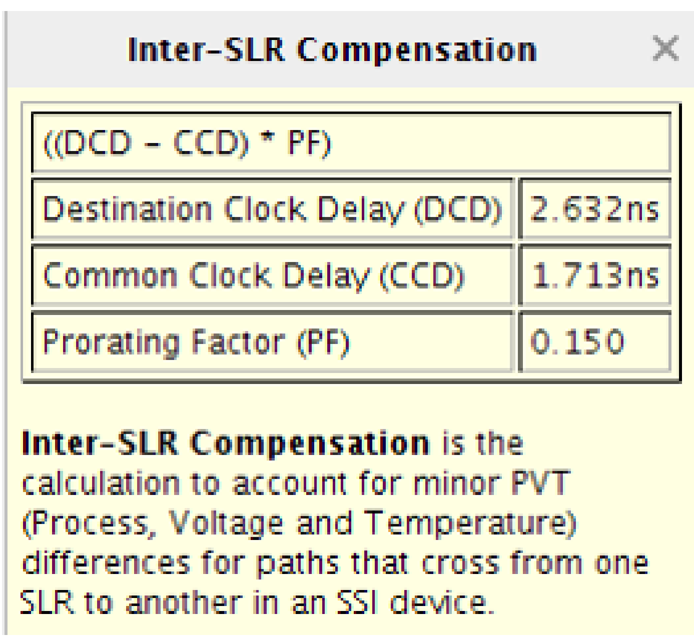


Fig. 10.1: Example Vivado tooltip window describing the Inter-SLR Compensation delay penalty

In RapidWright, we have created a parametrized, stand-alone application that can automatically generate a placed and routed DCP from scratch that implements a circuit that eliminates and minimizes the two challenges mentioned above. First, it creates a netlist with pairs of flops that are connected and placed and routed across SLR crossings using the dedicated Laguna TX/RX flip flop sites. Next, it custom routes the clock (the circuit has its own BUFGCE) such that it can individually tune the leaf clock buffers (LCBs) for each direction on each side of the SLR. By using the LCBs, the hold time in the first challenge mentioned above is eliminated. To minimize the ISC penalty, a clock root is generated for each clock region (CR) that contains an SLR crossing.

10.1.1 Steps to Run

1. Ensure you have RapidWright correctly setup and/or installed. See the *Getting Started* page for details.
2. Run `java com.xilinx.rapidwright.examples.SLRCrosserGenerator -h`. This will print all the available options to parameterize the SLR crossing output, example output below:

```
==============================================================================
==                        SLR Crossing DCP Generator                        ==
==============================================================================
This RapidWright program creates a placed and routed DCP that can be
imported into UltraScale+ designs to aid in high speed SLR crossings.  See
RapidWright documentation for more information.

Option                                    Description
------                                    -----------
-?, -h                                    Print Help
-a [String: Clk input net name]           (default: clk_in)
-b [String: Clock BUFGCE site name]       (default: BUFGCE_X0Y218)
-c [String: Clk net name]                 (default: clk)
-d [String: Design Name]                  (default: slr_crosser)
-i [String: Input bus name prefix]        (default: input)
-l [String: Comma separated list of       (default: LAGUNA_X2Y120)
  Laguna sites for each SLR crossing]
-n [String: North bus name suffix]        (default: _north)
-o [String: Output DCP File Name]         (default: slr_crosser.dcp)
-p [String: UltraScale+ Part Name]        (default: xcvu9p-flgc2104-2-i)
-q [String: Output bus name prefix]       (default: output)
-r [String: INT clk Laguna RX flops]      (default: GCLK_B_0_1)
-s [String: South bus name suffix]        (default: _south)
-t [String: INT clk Laguna TX flops]      (default: GCLK_B_0_0)
-u [String: Clk output net name]          (default: clk_out)
-v [Boolean: Print verbose output]        (default: true)
-w [Integer: SLR crossing bus width]      (default: 512)
-x [Double: Clk period constraint (ns)]   (default: 1.538)
-y [String: BUFGCE cell instance name]    (default: BUFGCE_inst)
-z [Boolean: Use common centroid]         (default: false)
```

3. A default scenario of a single bi-directional crossing of 512 bits is generated at the LAGUNA_X2Y120 site on a VU9P part if no options are provided. The DCP is generated in the current working directory with the name `slr_crosser.dcp` unless the `-o` option is specified.

```
$ java com.xilinx.rapidwright.examples.SLRCrosserGenerator
==============================================================================
==                           SLRCrosserGenerator                            ==
==============================================================================
                    Init:     4.787s
          Create Netlist:     0.123s
     Place SLR Crossings:     0.121s
```

```
      Custom Clock Route:      3.756s
           Route VCC/GND:      0.079s
              Write EDIF:      0.148s
     Writing XDEF Header:      0.090s
  Writing XDEF Placement:      0.213s
    Writing XDEF Routing:      0.404s
 Writing XDEF Finalizing:      0.079s
             Writing XDC:      0.039s
------------------------------------------------------------------------------
          [No GC] *Total*:      9.839s
Wrote final DCP: /home/user/slr_crosser.dcp
```

4. Open the DCP using Vivado to view the design. It should look similar to the annotated screenshot below:

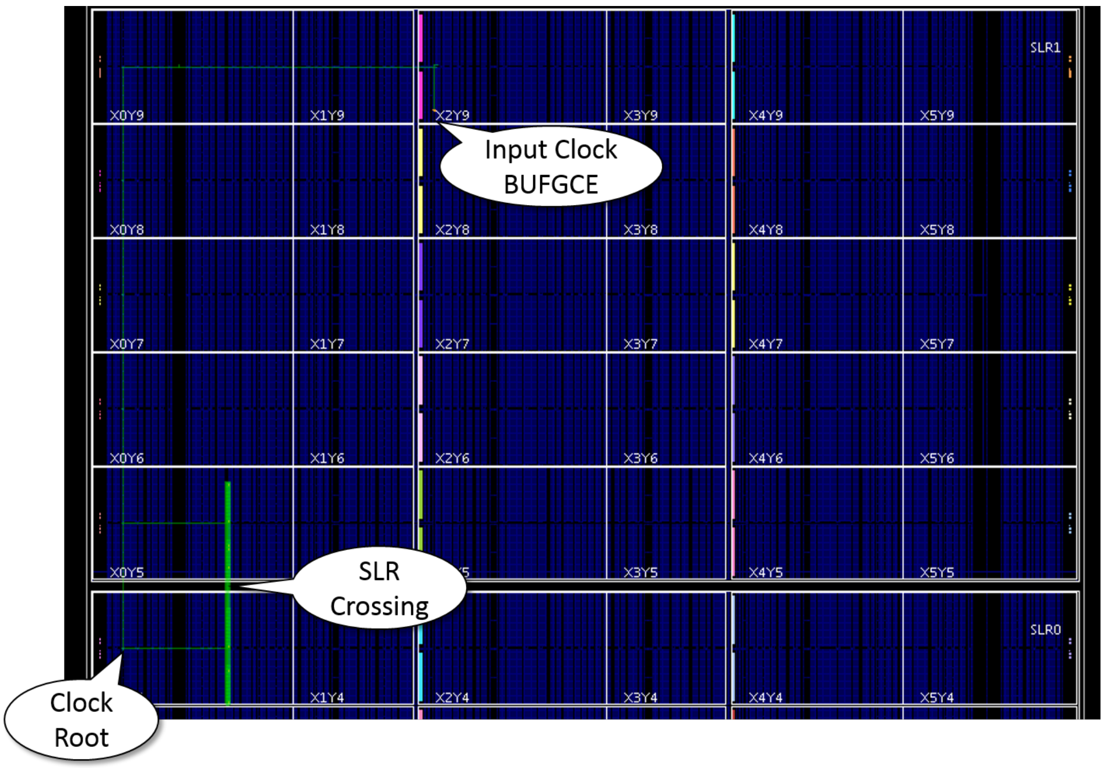


Fig. 10.2: Vivado Screenshot with bubble annotations of a single, bi-direction 512-bit SLR crossing circuit.

5. You can also unzip the DCP (treating it like an ordinary ZIP file) and inside you'll find Verilog and VHDL stubs that can be imported into RTL designs for black box inclusion. Example output below:

```
$ unzip slr_crosser.dcp
Archive:  slr_crosser.dcp
  inflating: slr_crosser.edf
  inflating: slr_crosser.xdef
  inflating: slr_crosser_late.xdc
  inflating: slr_crosser_stub.v
  inflating: slr_crosser_stub.vhdl
  inflating: dcp.xml
$ cat slr_crosser_stub.v
// This file was generated by RapidWright 2018.2.0.
```

```
// This empty module with port declaration file causes synthesis tools to infer a␣
↪black box for IP.
// Please paste the declaration into a Verilog source file or add the file as an␣
↪additional source.
module slr_crosser(clk_in, clk_out, input0_north, input0_south, output0_north,␣
↪output0_south);
  input clk_in;
  output clk_out;
  input [511:0]input0_north;
  input [511:0]input0_south;
  output [511:0]output0_north;
  output [511:0]output0_south;
endmodule
$
```

Optionally, you can open the DCP in Vivado and write out the netlist as EDIF, Verilog or VHDL to be packaged as an IP. The DCP can then be dropped into the IP cache later.

6. As one additional example, the generator is capable of using every SLL in the device. To generate such a DCP for a VU9P device, run:

```
$ java com.xilinx.rapidwright.examples.SLRCrosserGenerator -w 720 -l LAGUNA_X0Y120,
↪LAGUNA_X2Y120,LAGUNA_X4Y120,LAGUNA_X6Y120,LAGUNA_X8Y120,LAGUNA_X10Y120,LAGUNA_
↪X12Y120,LAGUNA_X14Y120,LAGUNA_X16Y120,LAGUNA_X18Y120,LAGUNA_X20Y120,LAGUNA_X22Y120,
↪LAGUNA_X0Y360,LAGUNA_X2Y360,LAGUNA_X4Y360,LAGUNA_X6Y360,LAGUNA_X8Y360,LAGUNA_
↪X10Y360,LAGUNA_X12Y360,LAGUNA_X14Y360,LAGUNA_X16Y360,LAGUNA_X18Y360,LAGUNA_X20Y360,
↪LAGUNA_X22Y360
```

The resultant DCP should look similar to the following in Vivado:

10.2 Build an IP Integrator Design with Pre-Implemented Blocks

What You'll Need to Get Started:

- Vivado 2018.2 or later
- RapidWright 2018.2.2 or later

Note: This tutorial uses the Tcl script `rapidwright.tcl` found in RapidWright. If you are using a standalone jar, you can extract the rapidwright.tcl (and other device/data) by running `java -jar <standalone.jar> --unpack_data` and setting the environment variable RAPIDWRIGHT_PATH to the standalone jar location.

This tutorial will provides an example design and execution of the rapid prototyping flow found on the page *A Pre-implemented Module Flow*. It begins by creating an example MicroBlaze design in IP Integrator (IPI) and showing how RapidWright can pre-implement the blocks of the design, place them and route them. This tutorial is just a demonstration example and is still under development.

10.2.1 Procedure

0. Before running Vivado, be sure to make sure that your CLASSPATH environment variable is set properly (see *Manual Installation Steps*, steps 4 and 5). If using the standalone jar, just set CLASSPATH=<path_to_standalone_jar>.

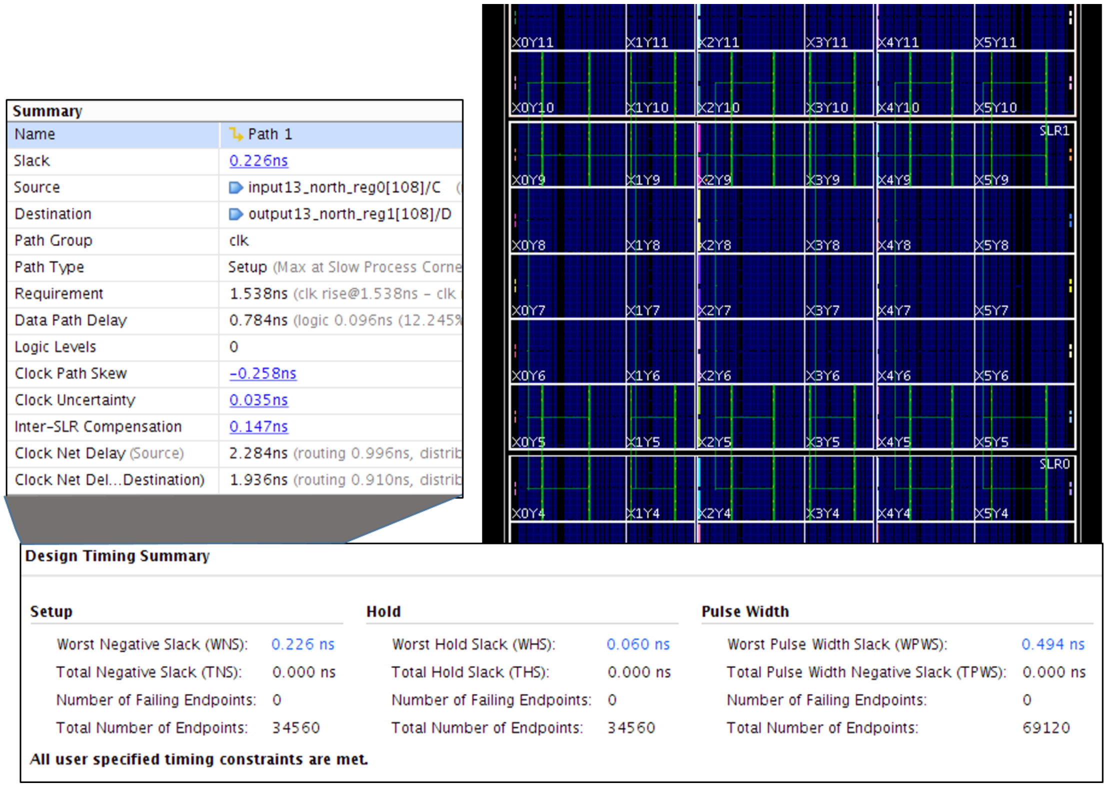


Fig. 10.3: Vivado Screenshot of all SLLs being used at potentially a 760MHz for a speed grade 2 device.

1. To generate an example placed and routed MicroBlaze design, run the following Tcl commands in your Vivado's Tcl prompt:

```
create_project project_1 [pwd]/testBlockStitcher -part xcku040-ffva1156-2-e
set_property board_part xilinx.com:kcu105:part0:1.4 [current_project]
set_property target_language VHDL [current_project]
create_bd_design "base_mb" -mode batch
update_compile_order -fileset sources_1
update_compile_order -fileset sim_1
instantiate_example_design -design base_mb base_mb
```

2. An XDC (Xilinx Design Constraints) file will be required for this design. Please download `base_mb.xdc` amd put it into the root of your project directory (`[pwd]/testBlockStitcher`). The XDC file is required for the RapidWright flow because IP Integrator and the OOC flow may not communicate all necessary information and the RapidWright block stitcher will need to instantiate most of the IOBs automatically.

3. To run the RapidWright pre-implemented block flow, you will need to source rapidwright.tcl, add the XDC from above to the project and run the flow with the following Tcl commands:

```
cd [pwd]/testBlockStitcher
# Download base_mb.xdc into this directory
add_files -fileset constrs_1 -norecurse [pwd]/base_mb.xdc
source ${env(RAPIDWRIGHT_PATH)}/tcl/rapidwright.tcl
rapid_compile_ipi
```

Note: If you have run this tutorial with previous versions of RapidWright, please backup your cache (if needed) run the RapidWright Tcl command `ultra_clear_cache` to reset the cache as some state in the cache is used differently.

This Tcl procedure will invoke Vivado to generate, synthesize, area constrain, place and route the various IPs out-of-context in the IPI design. Unless you set the environment variable `IP_CACHE_PATH`, the IP and pre-implemented block cache defaults to `$HOME/blockCache`. If the cache is empty, it will take several minutes to pre-implement each block. However, this process will eventually conclude with the design being fully stitched together in the `BlockStitcher` class in RapidWright and ultimately produce a fully placed (partially routed) DCP (the RapidWright router could be used to route the inter-block connections, but has a few outstanding issues so it is currently disabled).

4. Once the pre-implemented block flow is complete, you can test its recompilation speed by re-running:

```
rapid_compile_ipi
```

This should complete in less than a minute. Any connectivity changes to the design or adding blocks that have already been stored in the cache should always compile in less than a minute.

5. The stitched and placed RapidWright-produced DCP should be in the root directory of the project with the name `base_mb_placed.dcp`. This design can then be opened in Vivado by running the Tcl command:

```
open_checkpoint ./base_mb_placed.dcp
```

6. To finalize the implementation, `route_design` can be run to finish the design.

Note: The automatic block placer and router in RapidWright are still under development and although they can run quickly, their quality still needs to be improved. We recommend using the an implementation guide file (see *Implementation Guide File*). An IGF file directs the flow on how the blocks/IPs should be pre-implemented by specifying pblocks for the IPs and placement locations for the instances. The BlockStitcher will automatically generate an example IGF called `base_mb.igf.example` in root directory of the project.

More to come soon!

CHAPTER

ELEVEN

FREQUENTLY ASKED QUESTIONS

11.1 I can't open my DCP in RapidWright, I get 'ERROR: Couldn't determine a proper EDIF netlist to load with the DCP file . . . ', what should I do?

RapidWright is able to read any unencrypted design files. If a design/DCP has been encrypted, you'll need to generate a new file without encryption in order to use it with RapidWright.

However, sometimes without explicitly invoking encryption, Vivado will encrypt the EDIF file present in a DCP automatically (it is quite common). To enable reading the DCP within RapidWright, load the DCP in Vivado and then create a similarly named EDIF file (mydesign.dcp –> mydesign.edf) by running the command `write_edif mydesign.edf`. This will generate an unencrypted EDIF file (only if encryption is turned off and the design does not contain any encrypted IPs) that RapidWright can recognize and load in with the rest of the DCP.

RapidWright comes with a small utility called `ReplaceEDIFInDCP` that can avoid the use of two files for situations that may require that convenience.

11.2 Can RapidWright be used for designs targeting the AWS F1 platform?

Yes, there are some ways in which parts of a design generated in RapidWright can be inserted into an existing AWS-F1 design. One technique uses the Vivado command `read_checkpoint -cell <cell_instance_name> <checkpoint.dcp>`. If you insert a blackbox that matches your DCP (see the stub files inside the ZIP/DCP file) into your AWS-F1 design, you can use the `read_checkpoint` command to pull in a synthesized, placed and/or routed DCP into the existing design.

Note that RapidWright cannot read in the AWS F1 shell design as it is encrypted and user design data is encrypted by default.

11.3 When should I use RapidWright and when should I use Vivado?

We recommend that Vivado be used for all tasks that meet the users expectations. If you have designs that are running successfully and meeting your design constraints, there is no need to use RapidWright. However, if you are seeking to improve performance and/or productivity because of unique insights you might have into your application and/or the FPGA architecture being targeted, RapidWright might be able to help. Vivado will always be part of the flow for validating designs (DRC/Timing) and creating bitstreams. However, there may be strategic design structures that can be created, preserved and/or replicated in RapidWright that might help you achieve your performance goals.

11.4 What languages does RapidWright support, and how do I interact with them?

RapidWright is written in Java. RapidWright is also packaged with a Python interpreter called Jython that enables it to run pure Python scripts and code. We recommend that for more compute intensive work, Java implementations be the language of choice as it will execute faster. Python is especially useful for interacting with RapidWright in a command-line type fashion. This allows device and design objects to remain persistent as the user examines their work and choose to make changes on the fly.

11.5 Why is the framework called RapidWright?

The 'Rapid' portion is to indicate speed and efficiency. It also provides some resemblance from a previous generation framework called RapidSmith. The 'Wright' portion was a common surname in England and means maker or builder. RapidWright is a framework to help you quickly build designs for Vivado.

11.6 Can RapidWright generate bitstreams?

No. There is currently no bitstream information in RapidWright. Any designs will need to be put back into Vivado for DRC and bitstream generation.

11.7 Does RapidWright have device timing information?

No. Currently, there is no timing information provided in RapidWright. To run timing, use the `Design.writeCheckpoint()` command and load the design in Vivado to report timing.

11.8 Is there any published work on RapidWright?

Yes, we had a paper at FCCM 2018 (The 26th IEEE International Symposium on Field-Programmable Custom Computing Machines). A preprint copy of the paper is available here: `FCCM18-RapidWright.pdf`. The presentation slides are avilable here: `FCCM18-RapidWright-Presentation.pdf`.

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search