RapidWright Documentation

Release 2017.4-beta

Xilinx Research Labs

May 03, 2018

CONTENTS

1 Introduction **1**
- 1.1 What is RapidWright? . . . 1
- 1.2 Why RapidWright? . . . 1
- 1.3 What about RapidSmith? . . . 2
- 1.4 Vivado and RapidWright . . . 2

2 Getting Started **5**
- 2.1 Quick Start (Try it out) . . . 5
- 2.2 Full Installation (Development) . . . 6

3 RapidWright Eclipse Setup **9**
- 3.1 Step-by-Step Instructions . . . 9
- 3.2 Setup Eclipse with Existing Repo . . . 14

4 FPGA Architecture Basics **23**
- 4.1 What is an FPGA? . . . 23
- 4.2 CPU vs. FPGA . . . 23
- 4.3 Lookup Tables (LUTs) . . . 24
- 4.4 State Elements . . . 27
- 4.5 Carry Chains . . . 27
- 4.6 DSP Blocks . . . 27
- 4.7 Block RAMs . . . 27

5 Xilinx Architecture Terminology **29**
- 5.1 Element / BEL (Basic Element of Logic) . . . 29
- 5.2 Site . . . 31
- 5.3 Tile . . . 33
- 5.4 FSR (Fabric Sub Region or Clock Region) . . . 34
- 5.5 SLR (Super Logic Region) . . . 34
- 5.6 Device . . . 34

6 RapidWright Overview **35**
- 6.1 Device Package . . . 35
- 6.2 EDIF Package (Logical Netlist) . . . 38
- 6.3 Design Package (Physical Netlist) . . . 39

7 Design Checkpoints **43**
- 7.1 What is a Design Checkpoint? . . . 43
- 7.2 What is Inside a Design Checkpoint? . . . 43
- 7.3 Xilinx Design Language (XDL) vs. Design Checkpoint . . . 44
- 7.4 RapidWright and Design Checkpoint Files . . . 45

8 Implementation Basics **47**
8.1 Placement . . . 47
8.2 Routing . . . 48

9 XDD File Format **51**
9.1 Tile Patterns . . . 51
9.2 Node Templates . . . 52
9.3 Intent Codes . . . 55
9.4 Site Types . . . 56
9.5 Tile Types . . . 58
9.6 Tiles . . . 59
9.7 Clock Regions . . . 59

10 A Pre-implemented Module Flow **61**
10.1 Background and Flow Comparison . . . 61
10.2 High Performance Flow . . . 62
10.3 Rapid Prototyping Flow . . . 65

11 RapidWright Tutorials **67**
11.1 Insert an ILA (ChipScope) into a Routed DCP . . . 67
11.2 Build an IP Integrator Design with Pre-Implemented Blocks . . . 69
11.3 Create Placed and Routed DCP to Cross SLR . . . 70

12 Indices and tables **75**

CHAPTER

ONE

INTRODUCTION

Table of Contents

- *Introduction*
  - *What is RapidWright?*
  - *Why RapidWright?*
  - *What about RapidSmith?*
  - *Vivado and RapidWright*

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)
- Intermediate, human-readable placement and routing database files format
- 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

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. 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

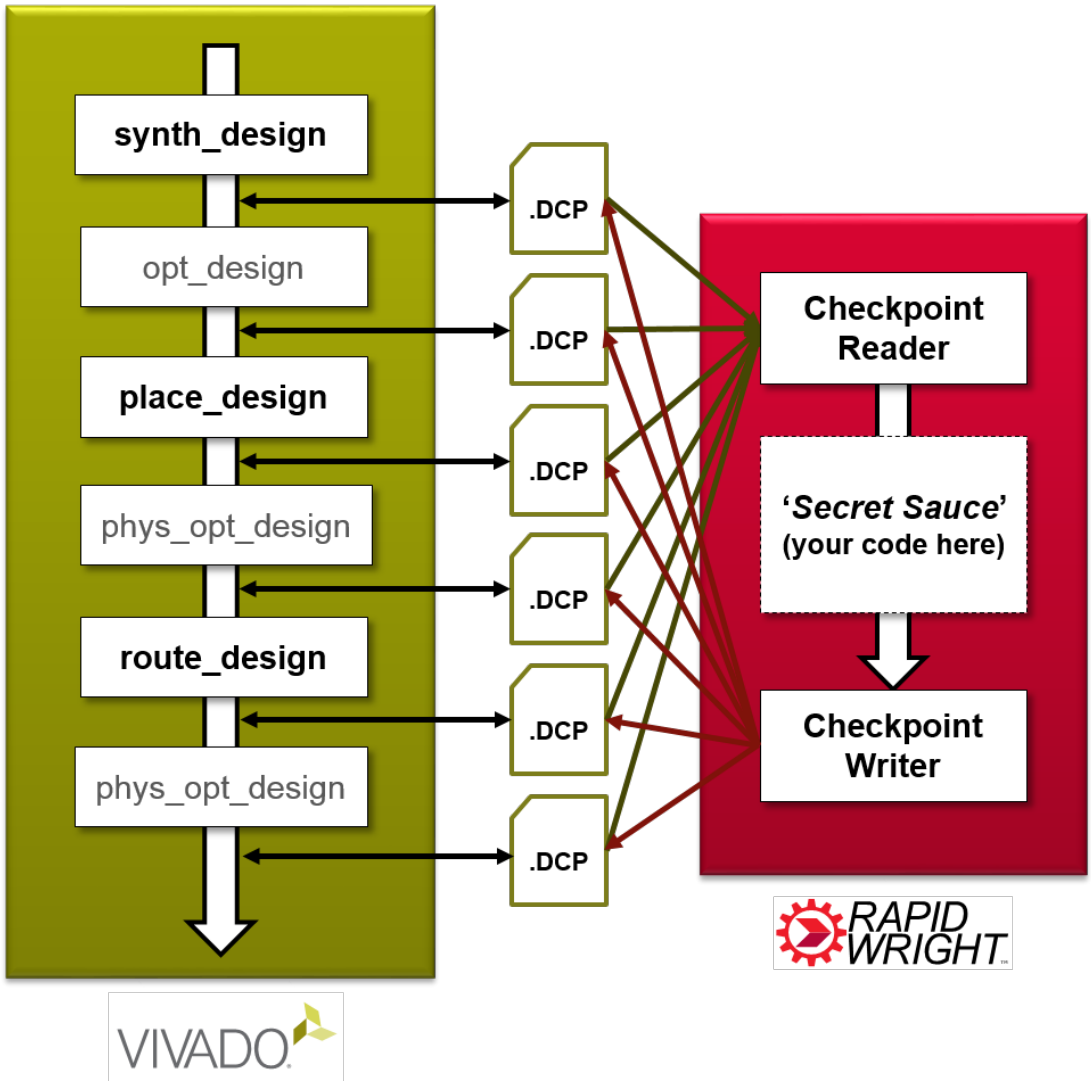
The Vivado Design Suite is the tool environment for developing and implementing designs for Xilinx FPGAs and SoCs. Vivado provides both a GUI environment and a Tcl scripting interface to control the various tools and steps involved in development. The Tcl scripting interface is quite powerful in that 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?

- *Getting Started*
  - *Quick Start (Try it out)*
  - *Full Installation (Development)*

2.1 Quick Start (Try it out)

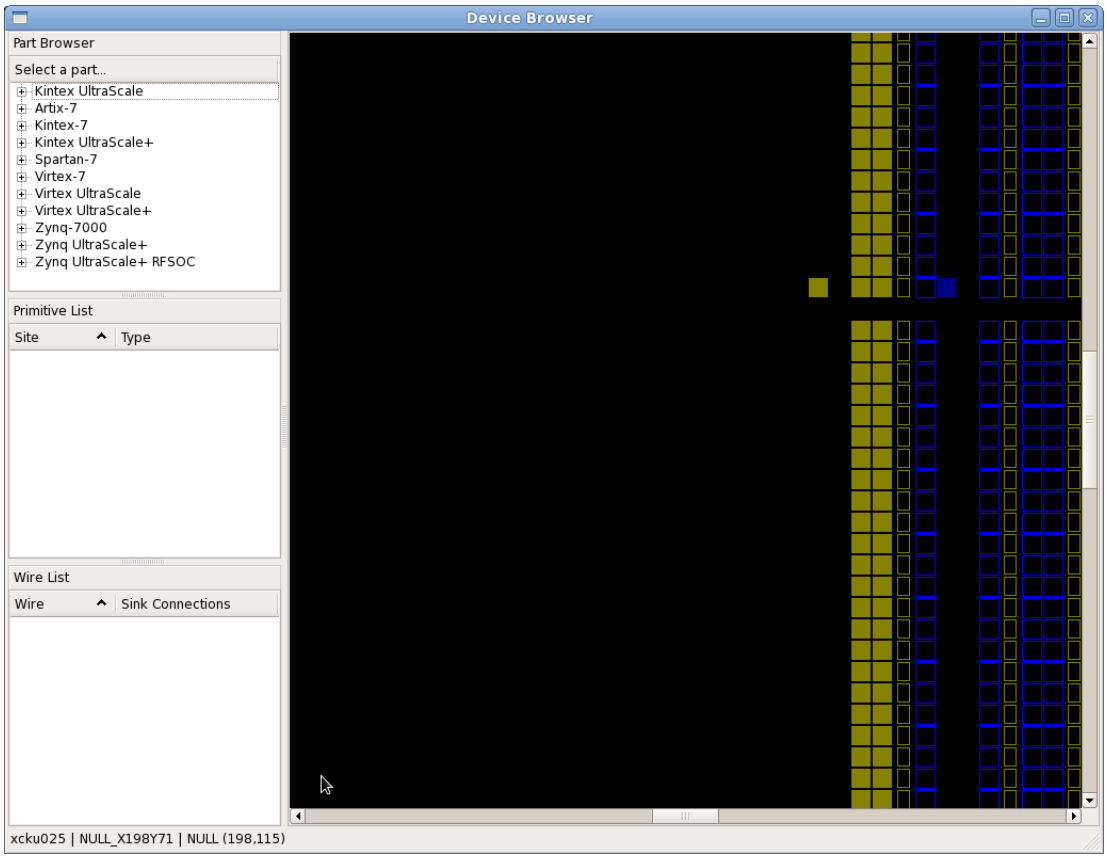
1. Download and install Oracle JRE/JDK Java 1.8 or later (install instructions here)
2. Download the latest standalone RapidWright release jar file
3. Start the RapidWright Python (Jython) interpreter by running:

```
$ java -jar rapidwright-2017.4-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 2017.4 (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

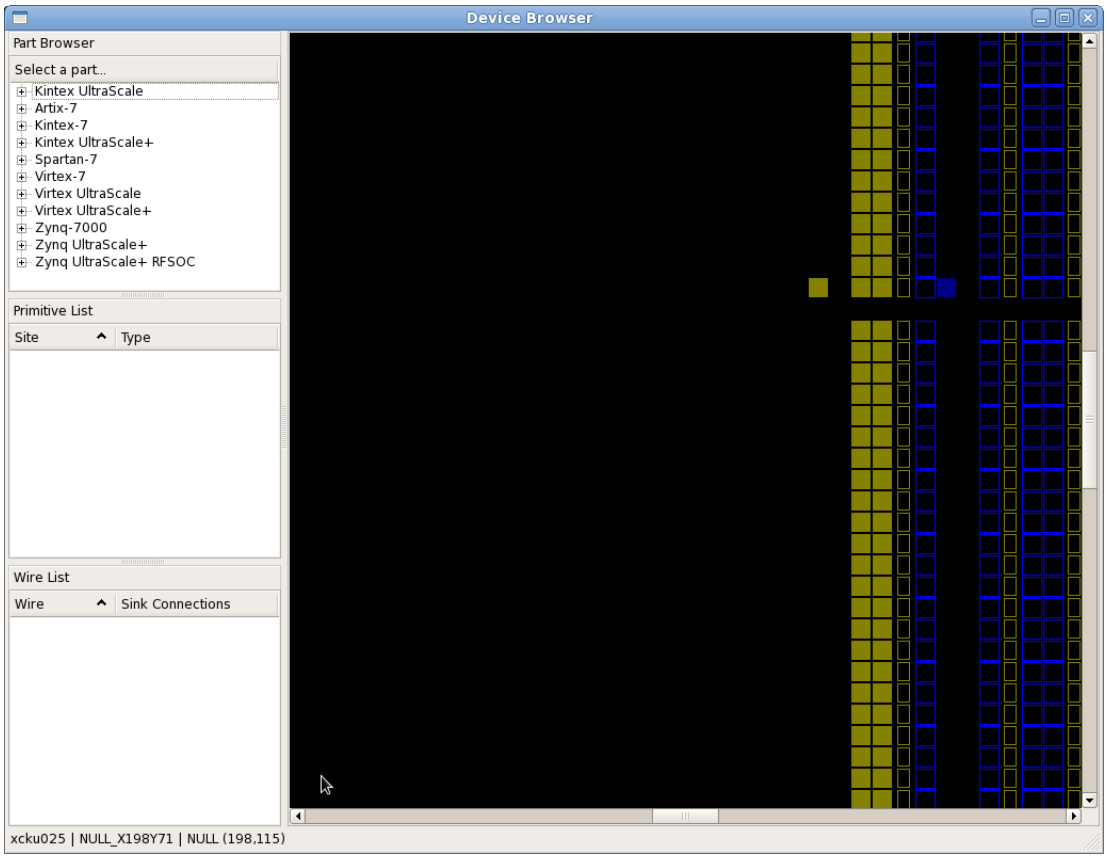
The easiest way to get RapidWright setup is to use the automatic installer jar that performs the manual installation automatically. Make sure you have the JDK and Git on your PATH.

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.

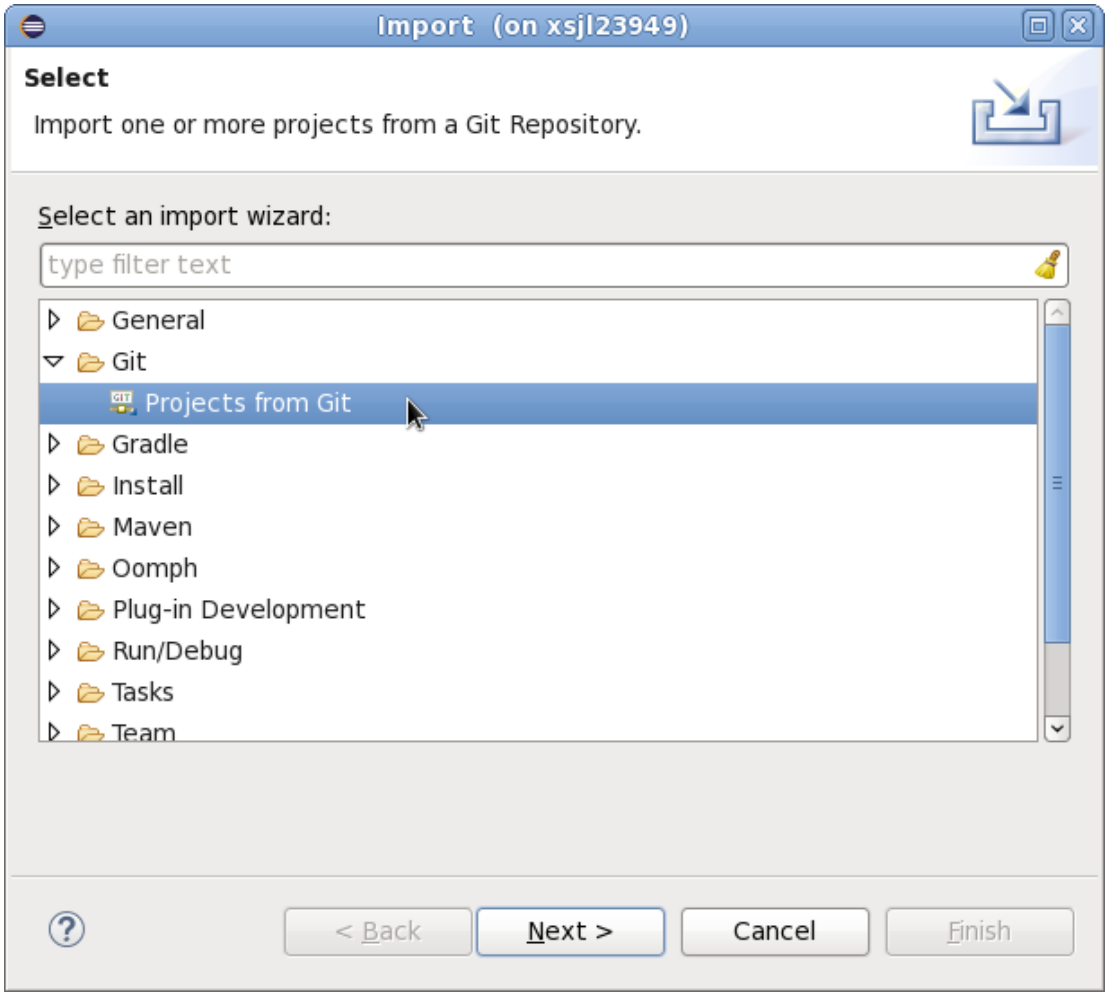
CHAPTER

THREE

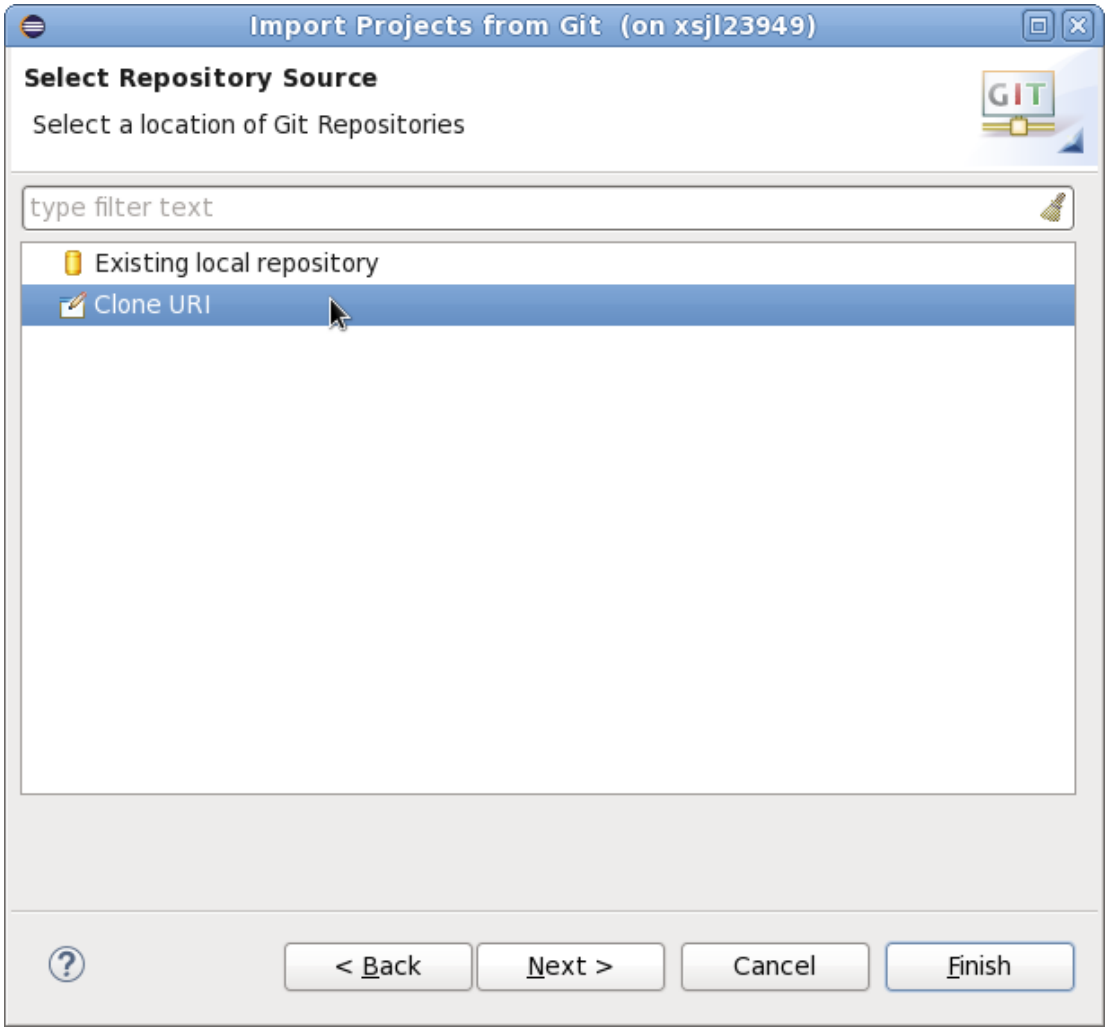
RAPIDWRIGHT ECLIPSE SETUP

3.1 Step-by-Step Instructions

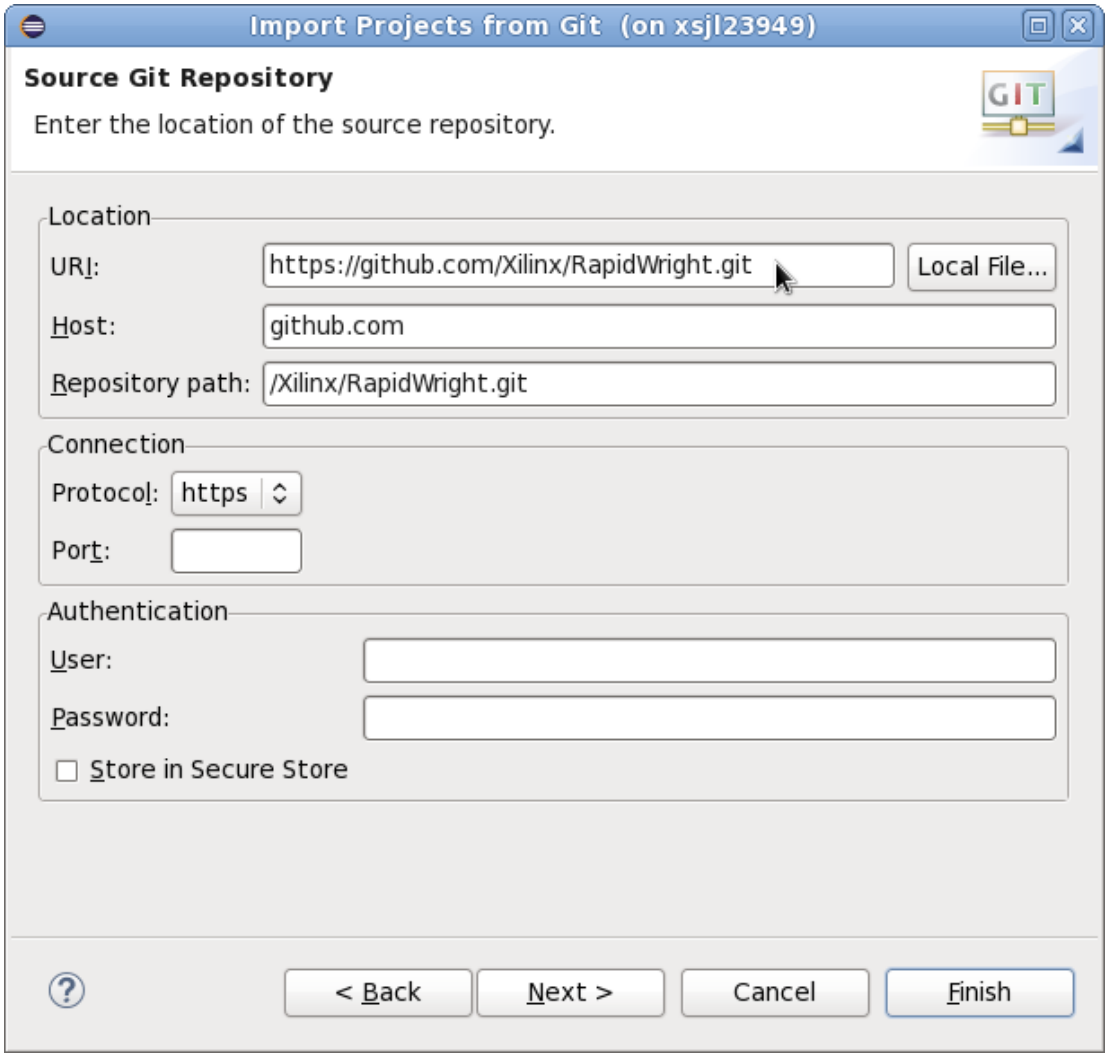
1. Make sure you have Java JDK 1.8 (or later) installed: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html Follow the instructions when running the downloaded executable. Add the `$(YOUR_JDK_INSTALL_LOCATION)/jdk1.x.x_x/bin` folder to your `PATH` environment variable.
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):



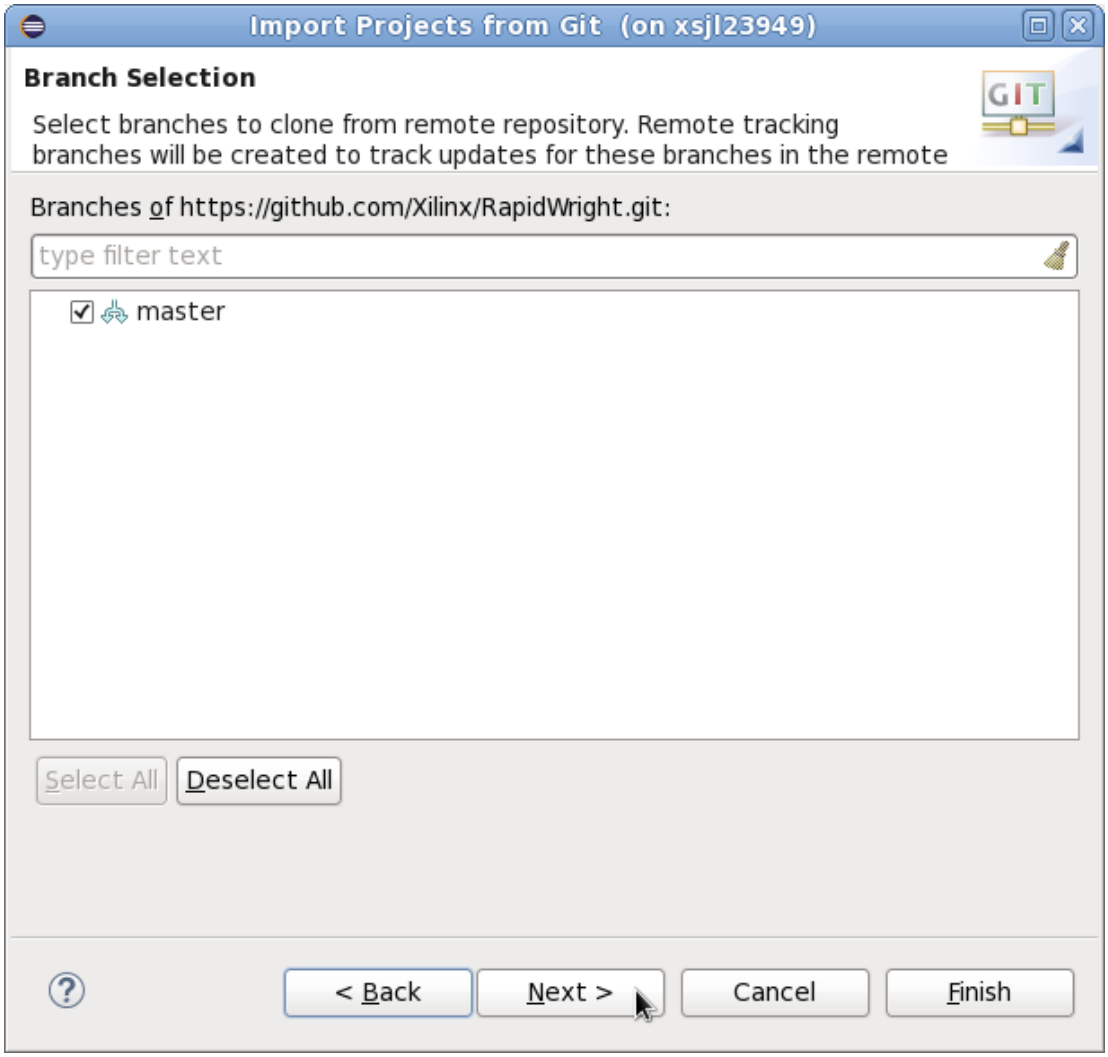
6. Choose Clone URI and click Next:



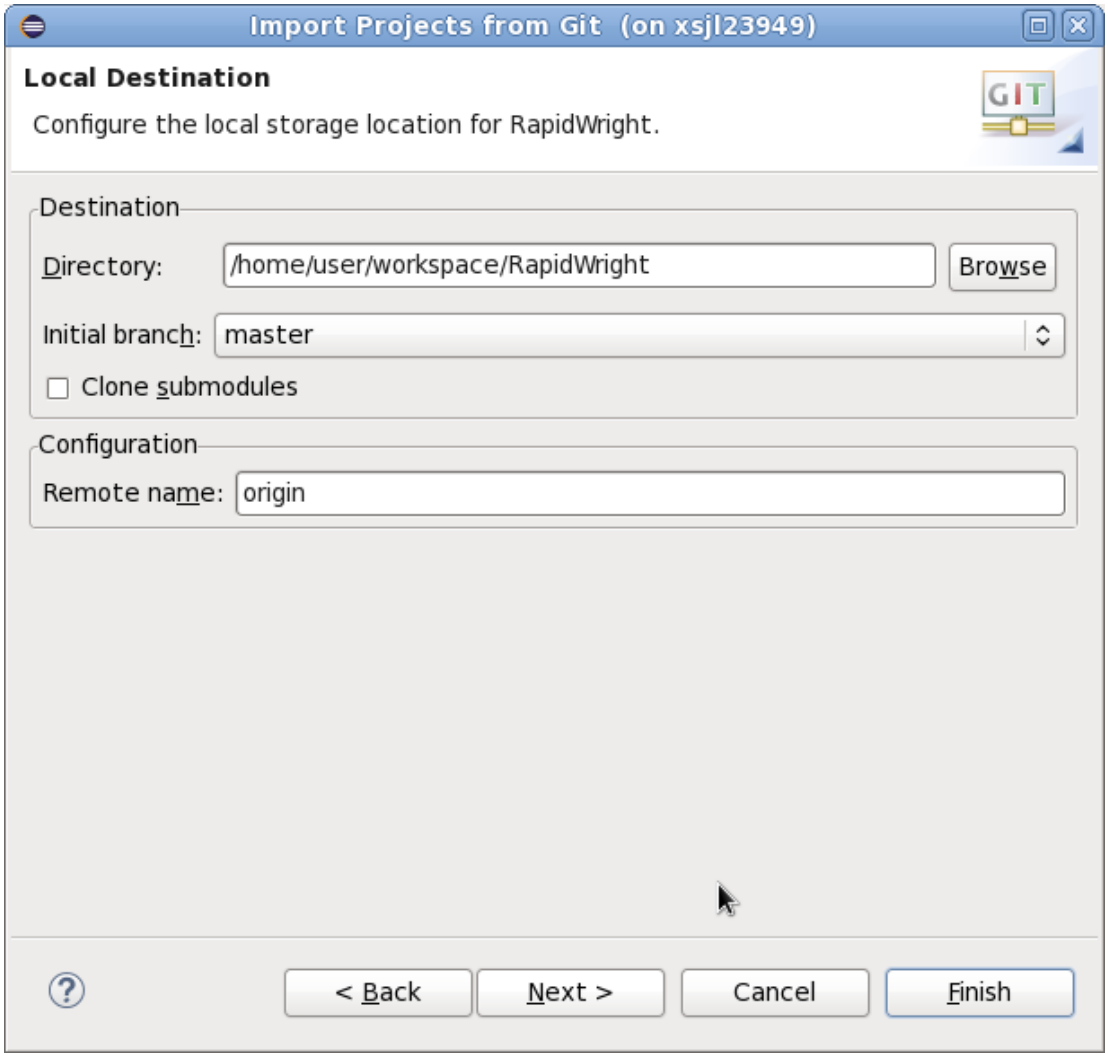
7. Copy and paste `https://github.com/Xilinx/RapidWright.git` into the URI box as shown below. The Host and Repository path fields should automatically be populated. Enter user and password (if applicable).



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.

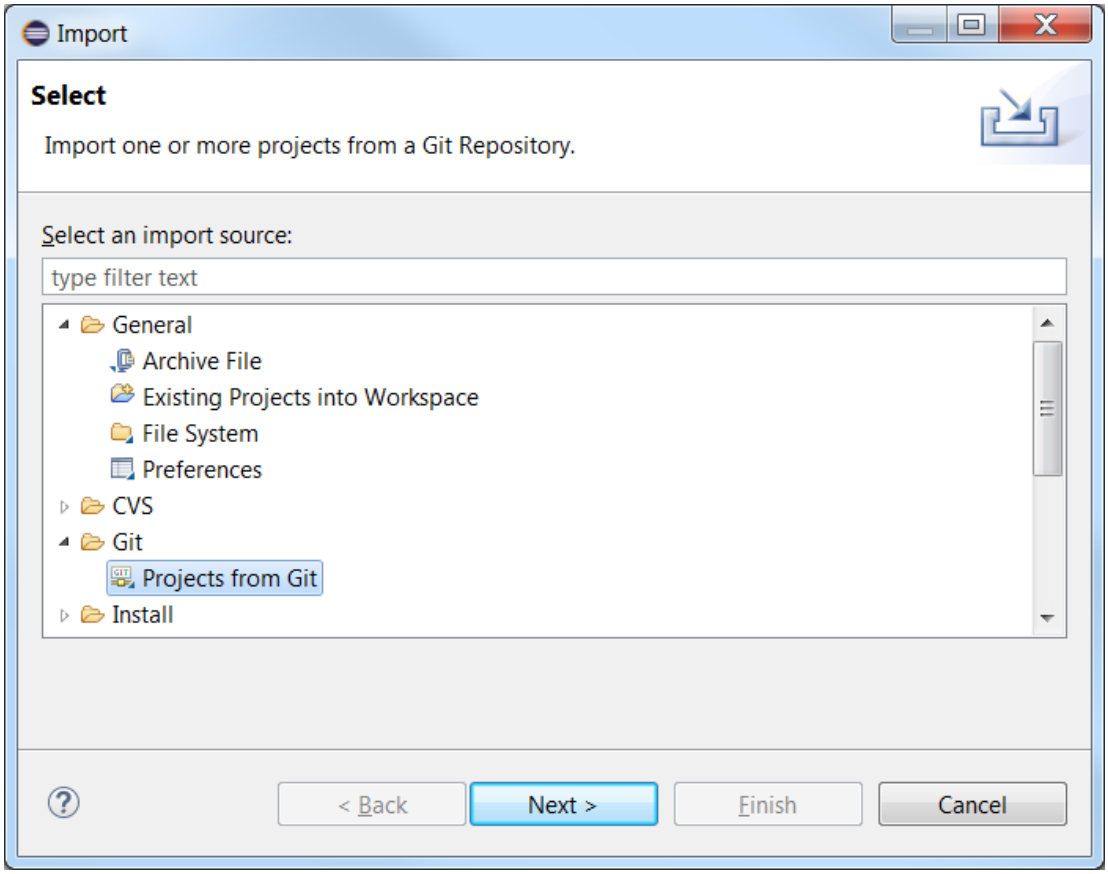


10. Continue with step 2 back on the *Manual Installation Steps* section of the *Getting Started* page.

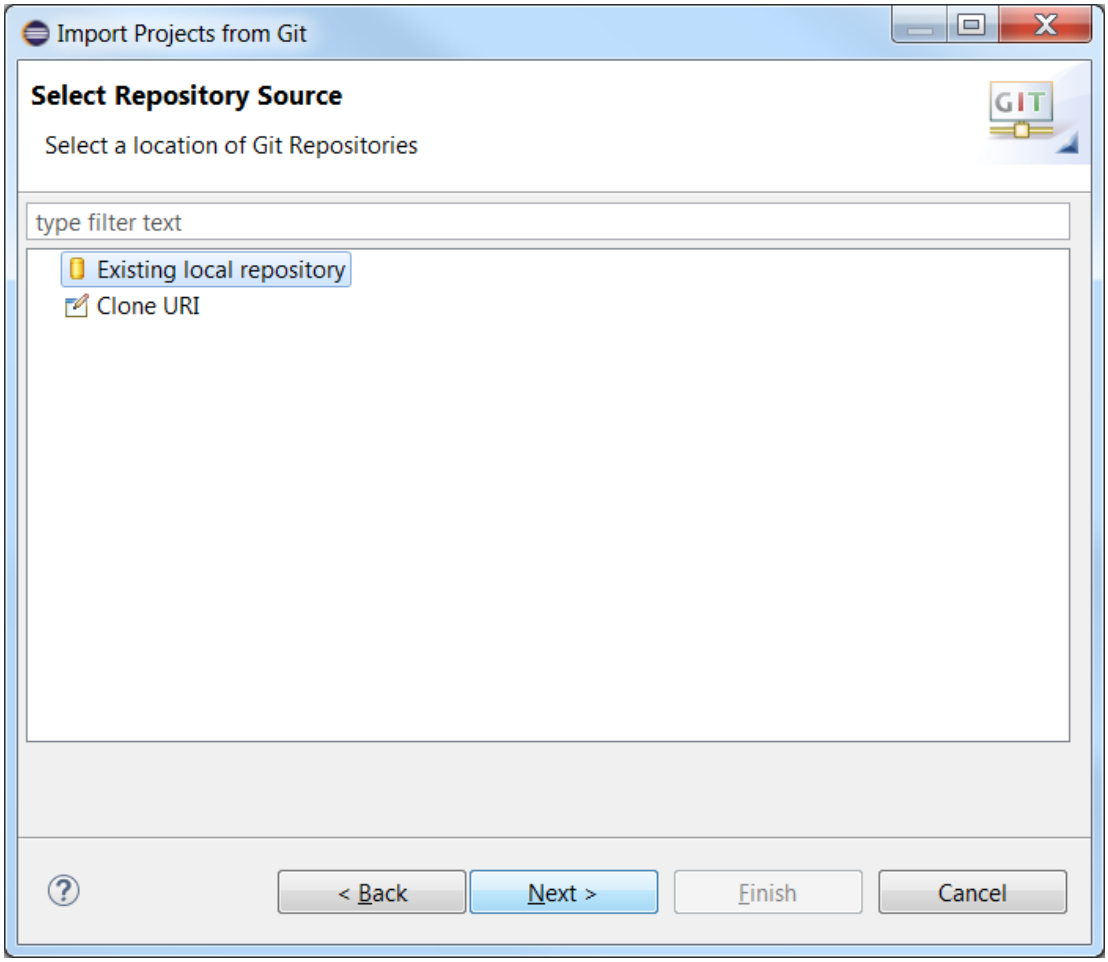
3.2 Setup Eclipse with Existing Repo

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)

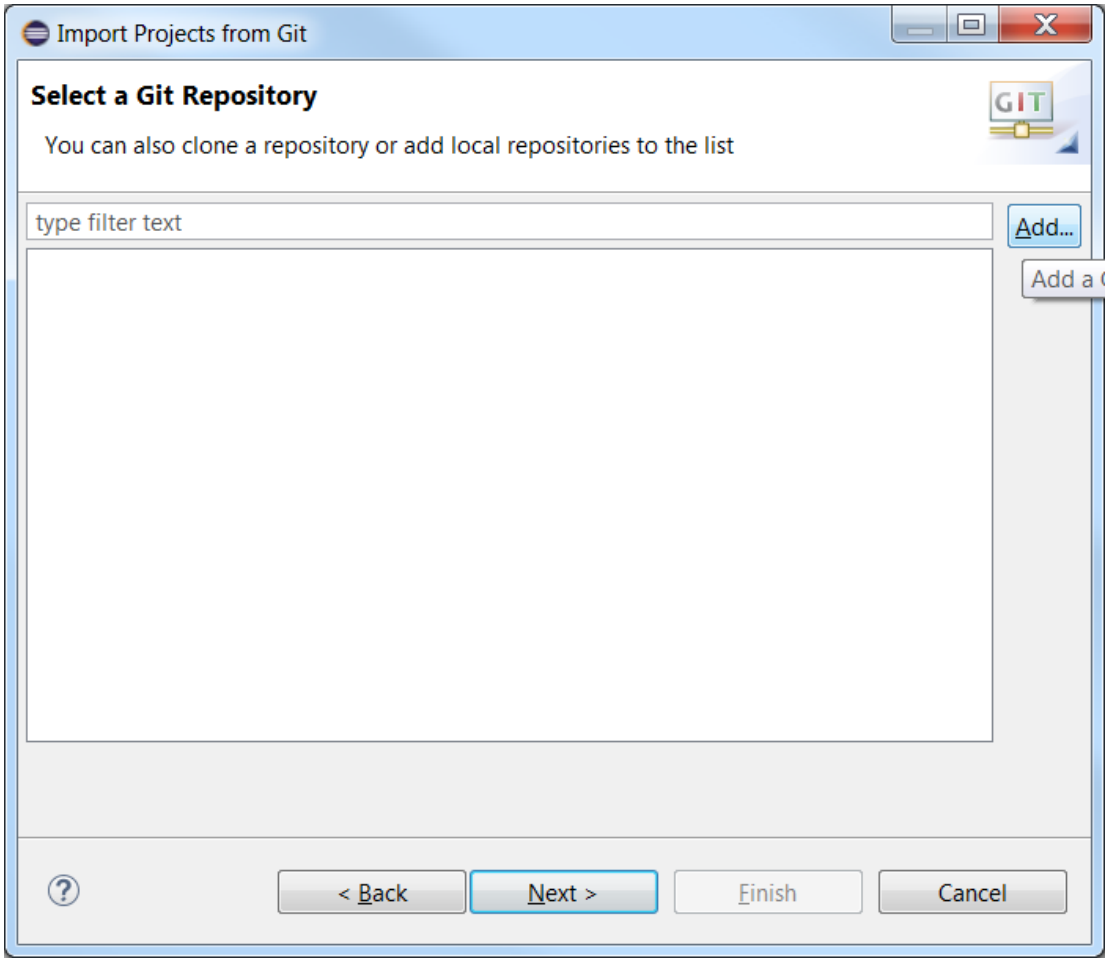
1. Make sure you have Java JDK 1.8 (or later) installed: http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html Follow the instructions when running the downloaded executable. Add the `$(YOUR_JDK_INSTALL_LOCATION)/jdk1.x.x_x/bin` folder to your `PATH` environment variable.
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):



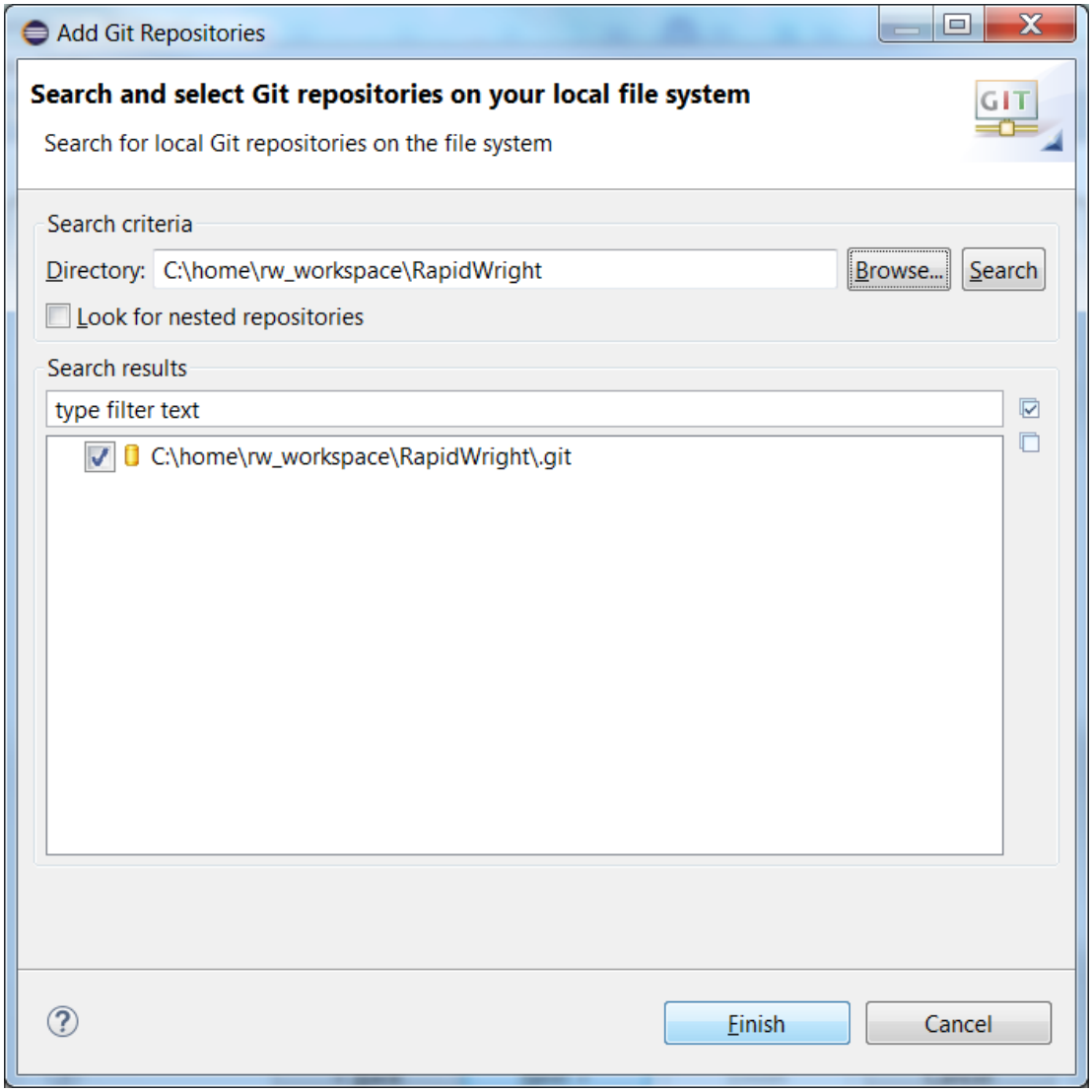
6. Choose ‘Existing local repository’, then click Next



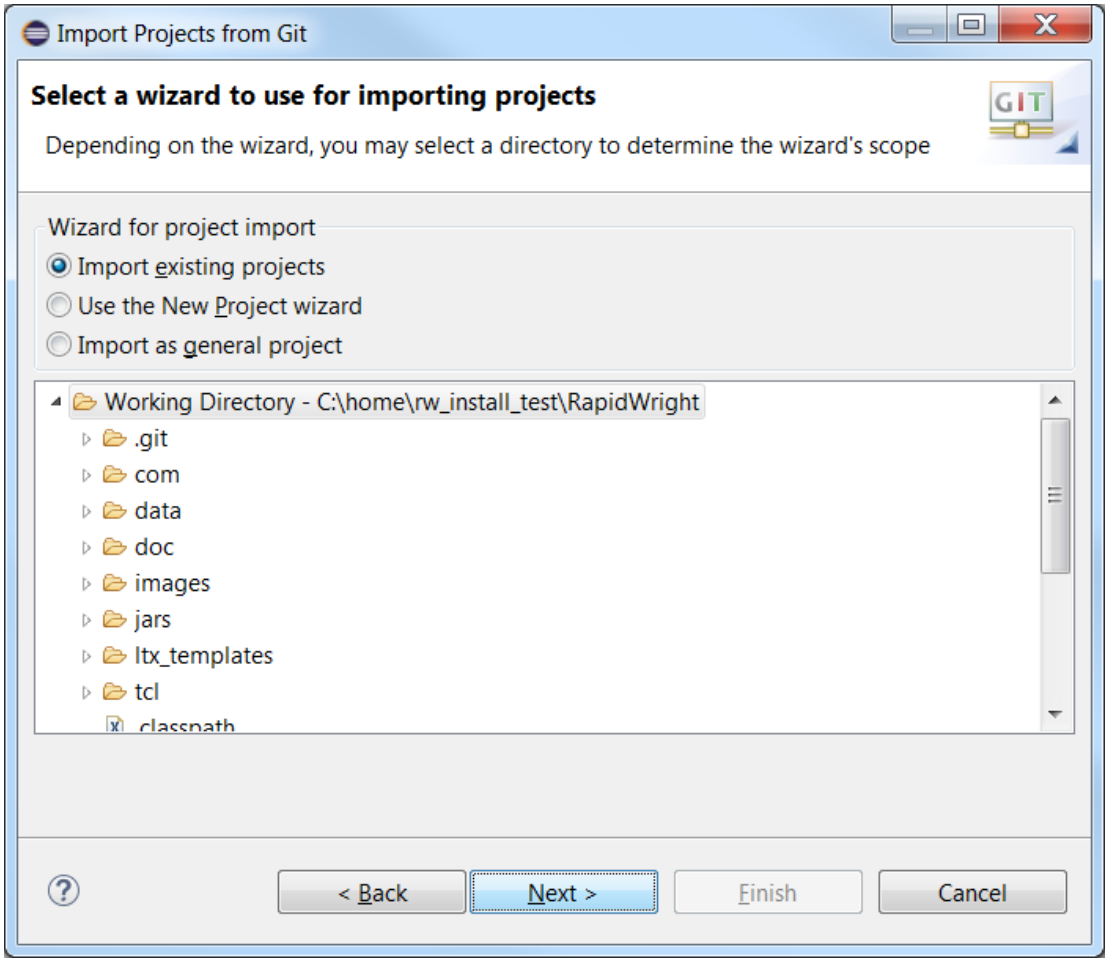
7. Select the existing repository by clicking the 'Add...' button



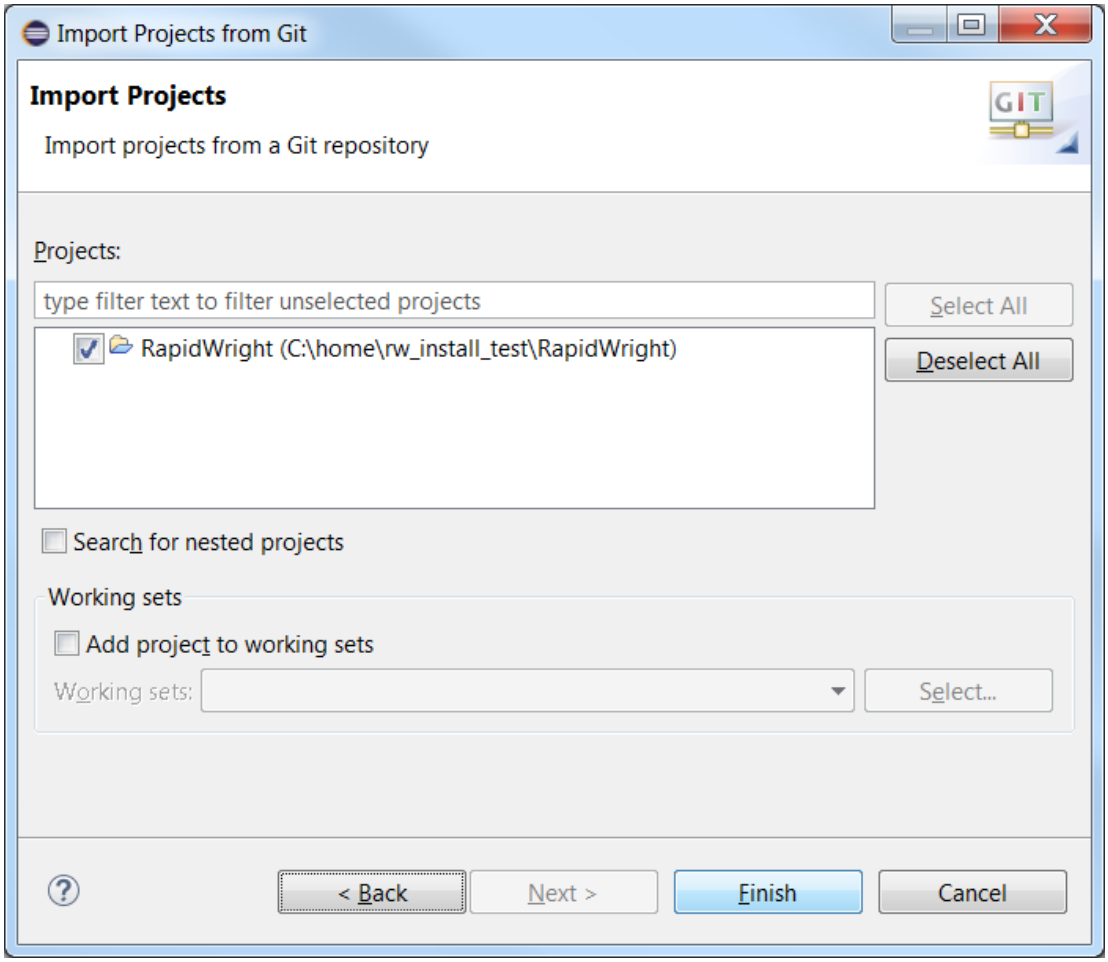
8. Enter the location of the repository in the 'Directory:' text box, check the box next to the name of the repo once it appears in the lower window. Click 'Finish' and then 'Next' on the previous window.



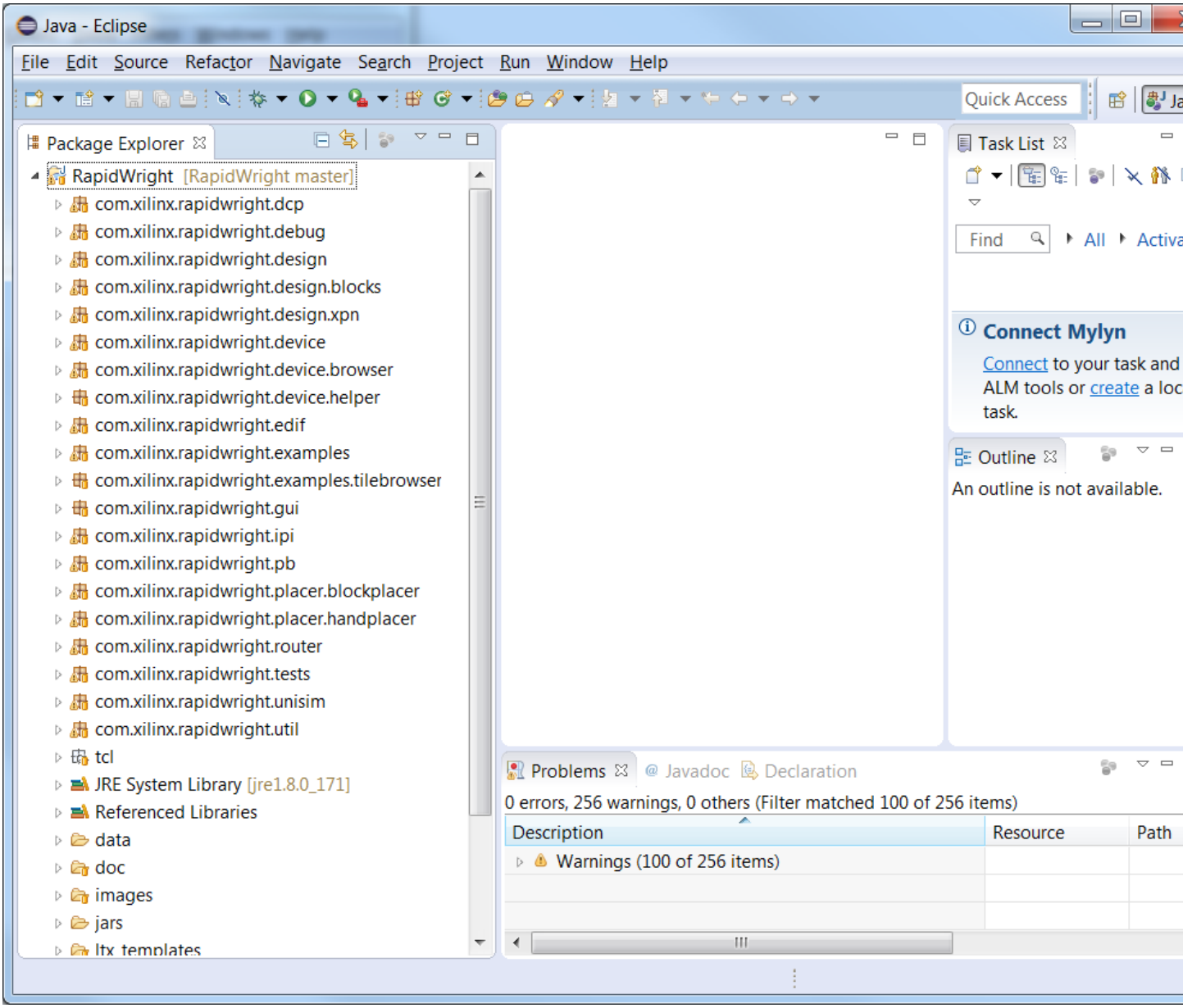
9. On the Wizard selection window, choose 'Import existing projects'. Then, click 'Next'.



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
ltx_templates
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

- *FPGA Architecture Basics*
  - *What is an FPGA?*
  - *CPU vs. FPGA*
  - *Lookup Tables (LUTs)*
  - *State Elements*
  - *Carry Chains*
  - *DSP Blocks*
  - *Block RAMs*

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

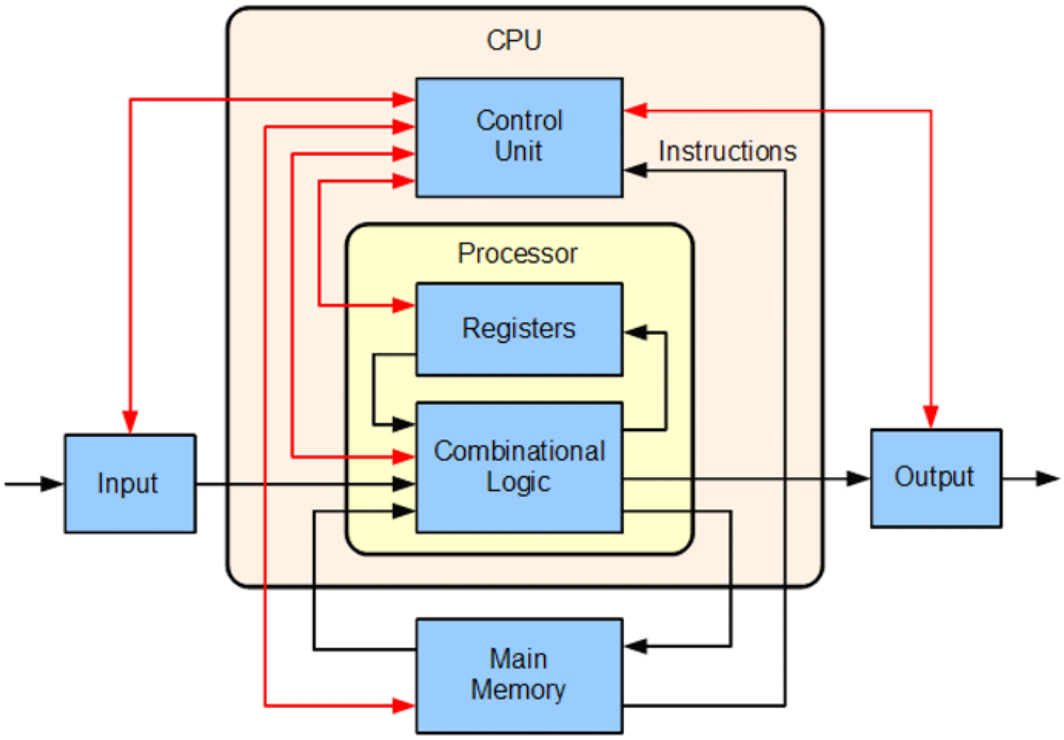


Fig. 4.1: Basic Von Neumann Processing Model for CPUs (Source: Labtron, Creative Commons).

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.

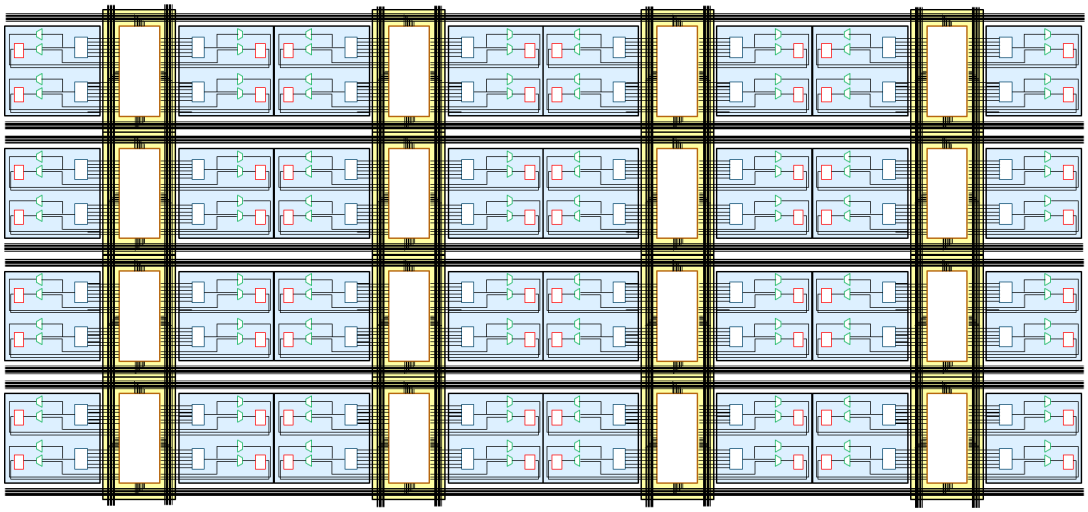


Fig. 4.2: Hypothetical FPGA logic array of LUTs, flip flops and programmable wires (interconnect)

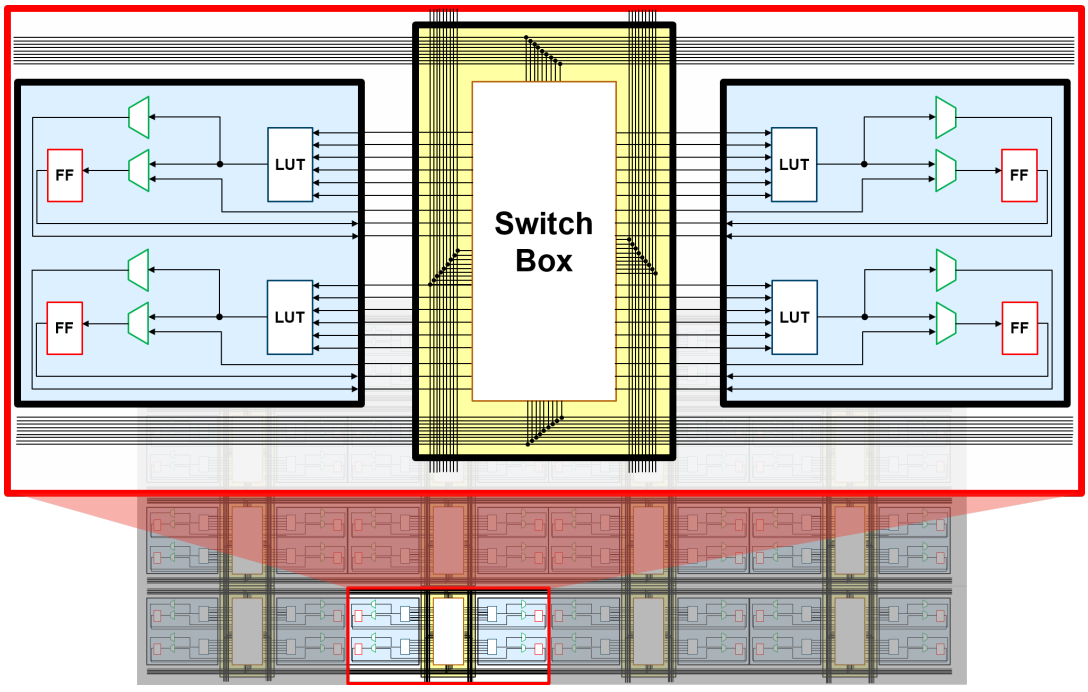
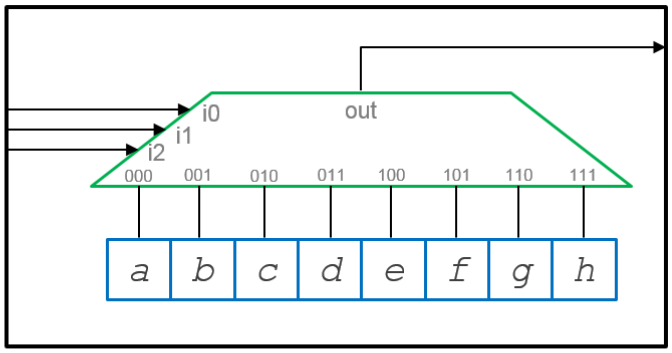


Fig. 4.3: Close up view of replicated tiles of the logic array and interconnect

i2	i1	i0	out
0	0	0	a
0	0	1	b
0	1	0	c
0	1	1	d
1	0	0	e
1	0	1	f
1	1	0	g
1	1	1	h

(a)



(b)

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

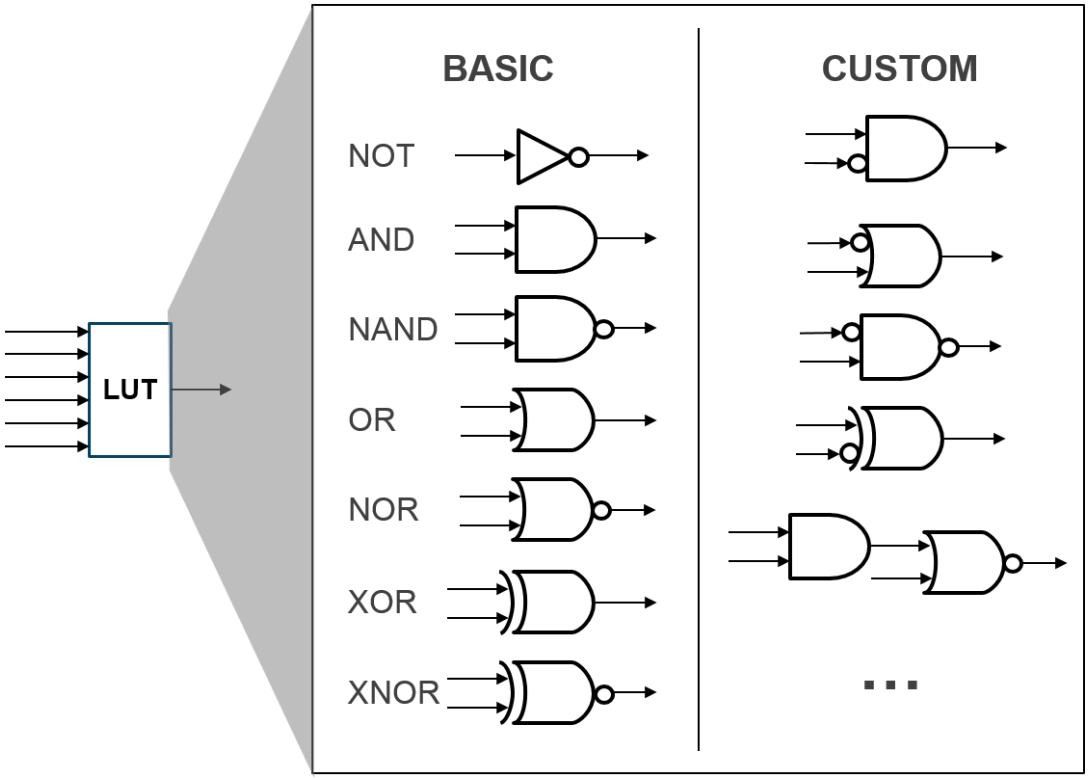


Fig. 4.5: Examples of several (but not all) logic functions a LUT can potentially implement

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

- *Xilinx Architecture Terminology*
  - *Element / BEL (Basic Element of Logic)*
  - *Site*
  - *Tile*
  - *FSR (Fabric Sub Region or Clock Region)*
  - *SLR (Super Logic Region)*
  - *Device*

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 Element / BEL (Basic Element of Logic)

At the lowest level, the atomic unit of Xilinx FPGAs is an Element. Elements are the smallest, indivisible, representable component in the fabric of an FPGA. There are two kinds of Elements, 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.

Elements have input and output pins. Elements also have configurable connections that connect an input pin to an output pin. These element-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

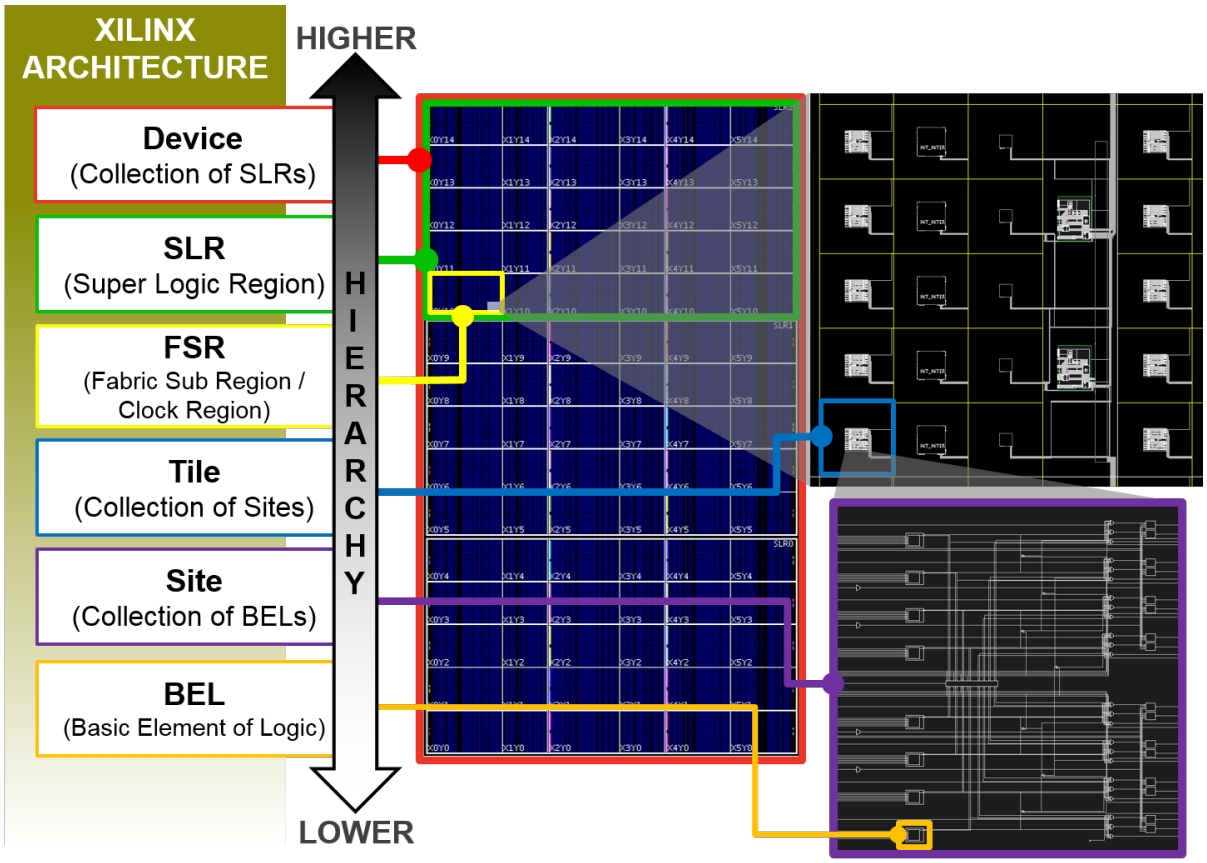


Fig. 5.1: Levels of architectural hierarchy in Xilinx FPGAs.

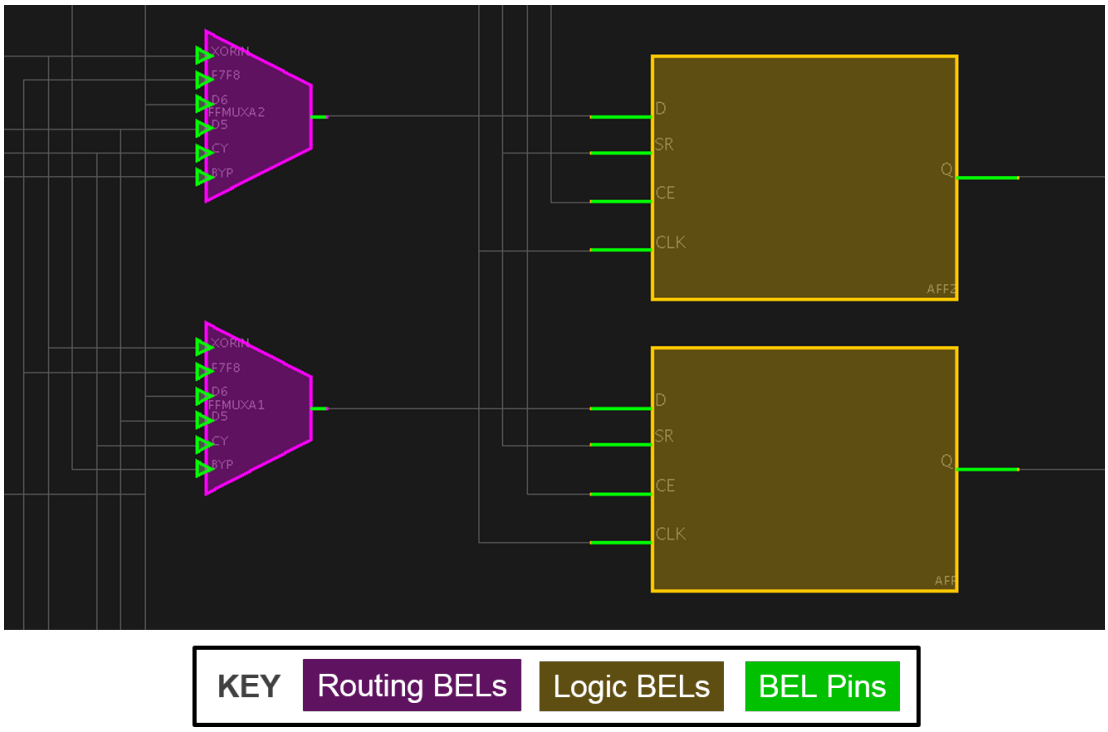


Fig. 5.2: Vivado representation of two routing muxes (routing BELs) and two flip flops (logic BELs).

in logic 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. Elements (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.

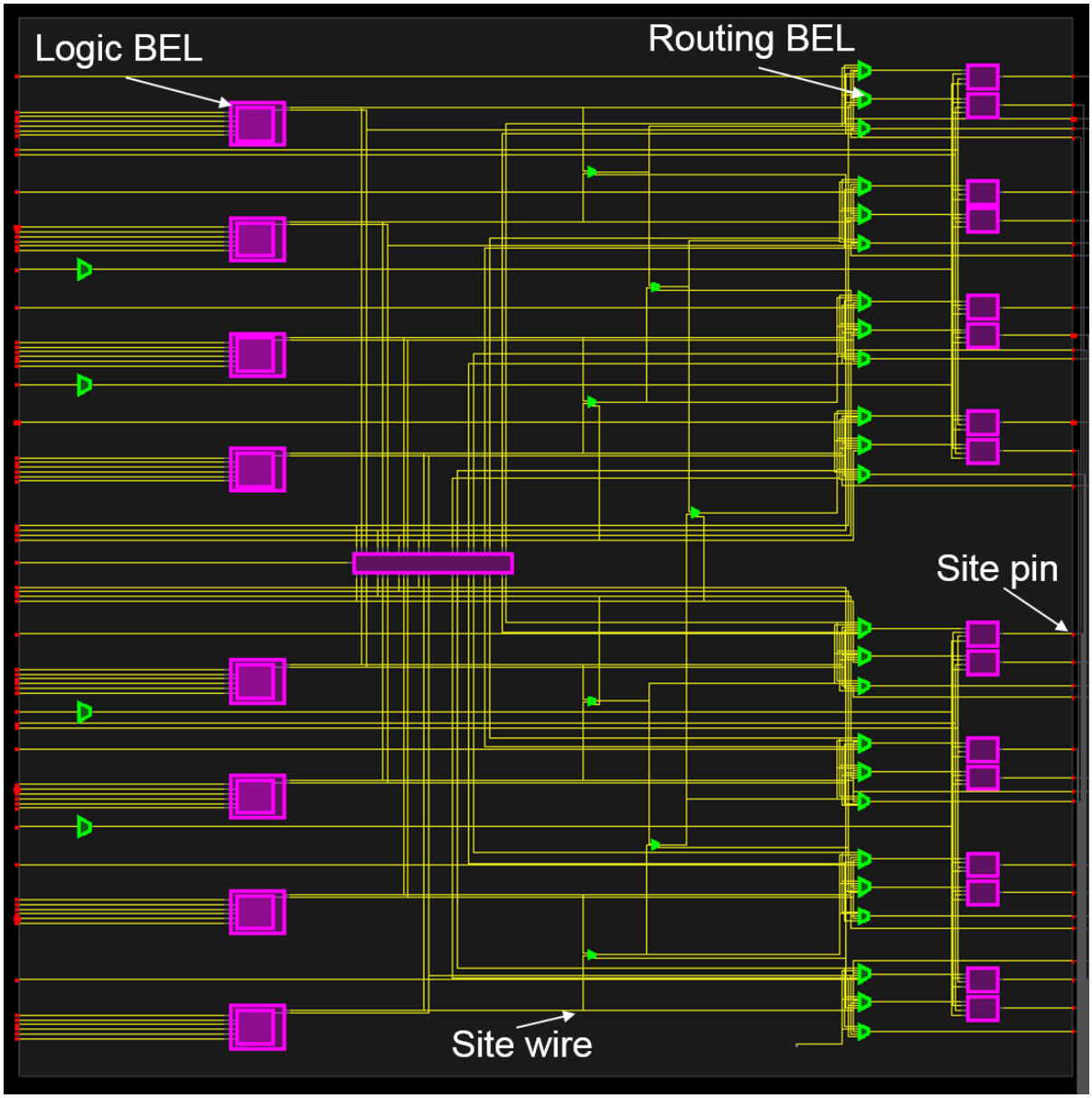
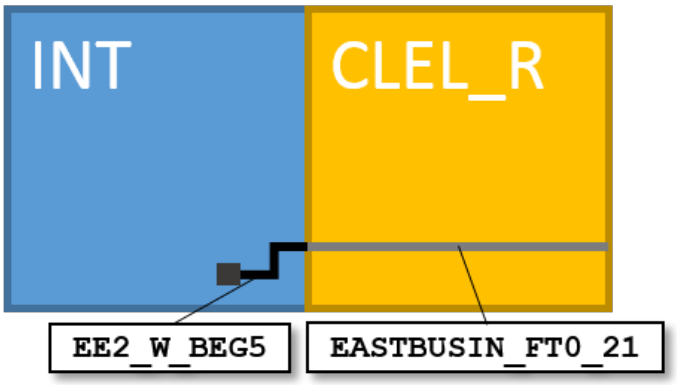


Fig. 5.3: An UltraScale+ SLICEL site, where logic BELs are magenta, routing BELs are green, site pins are red and site wires are yellow.

5.3 Tile

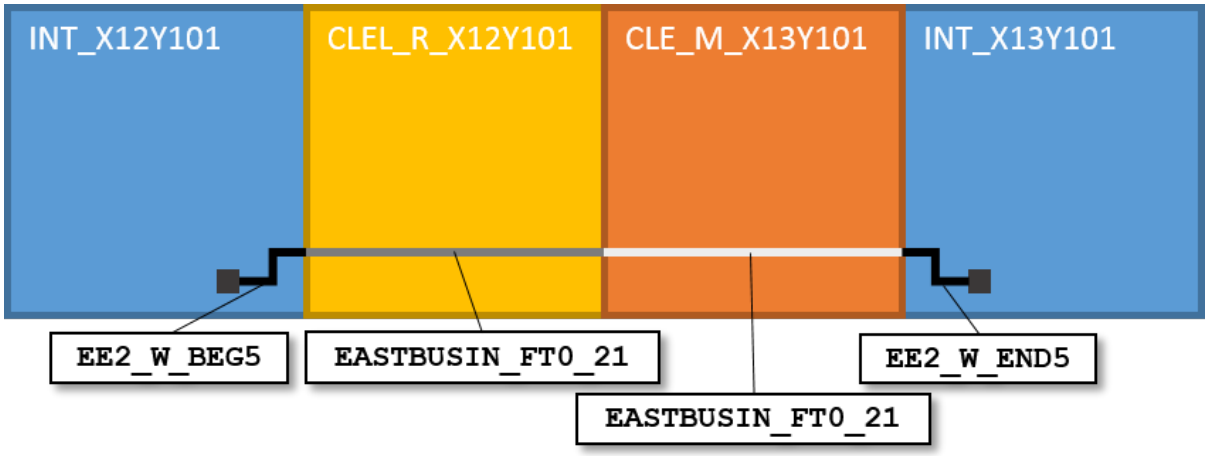
At an abstract level, Xilinx devices are created by assembling a grid of tiles. Similar to sites, each tile is an instance of a type and each tile has a unique name with an `_X#Y#` suffix. Tiles are the building blocks used when constructing an FPGA device. Tiles are designed to abut one another when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

Not all tiles contain sites and those that do, can have more than one. Unlike sites and elements, 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

- *RapidWright Overview*
  - *Device Package*
  - *EDIF Package (Logical Netlist)*
  - *Design Package (Physical Netlist)*

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. However, to get a textual representation of the data stored in a device file, you can use the `XDDPrinter` class to generate a Xilinx Device Database (.xdd) file. This file format was created

specifically for RapidWright and potentially opens the door for experimenting with devices by modifying the XDD file and compiling a new device.

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.

In representing the device architecture, the main components can be loosely categorized into two main categories: definition objects and instance objects as shown in the figure below.

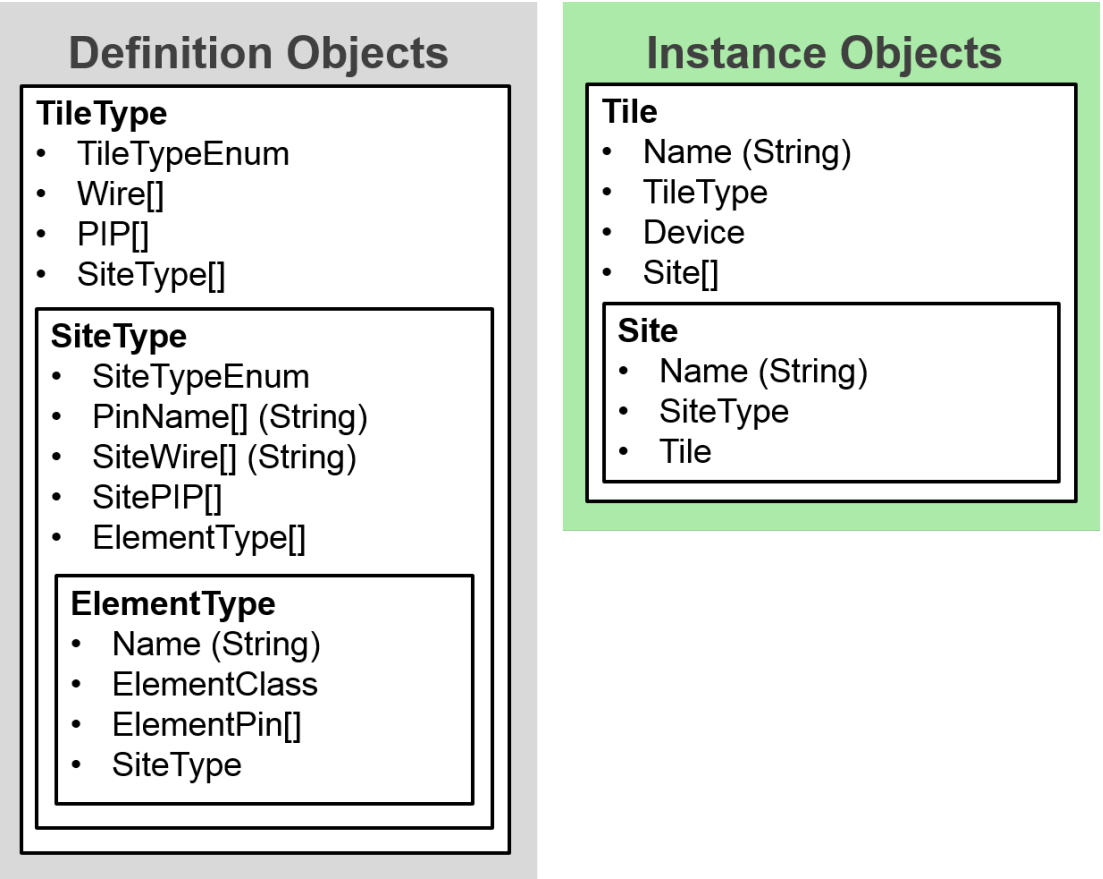


Fig. 6.1: Categorization of the core architectural classes with their most significant class member variables listed

Each `Device` object maintains a unique list of the definition objects `TileType` and `SiteType` that are referenced by `Tile` and `Site` instances objects, respectively. Note that the `ElementType` is unique among the other class types. It is a definition object that belongs to `SiteType` and there is no explicit instance object that elaborates it within the device architecture. This is because the collection of `ElementType` objects as defined in a `SiteType` is always unique. For example, in a `SiteTypeEnum.SLICEL SiteType`, there are 8 LUTs and 16 flip flops, but each is a uniquely named `ElementType`, distinguished by its name and context within the `SiteType` as shown in the figure below:

The `ElementType` class can be one of three kinds of non-routing objects in a `SiteType`: a Logic BEL, a Routing BEL and a Port (port of the SiteType). This is designated by its class member enum of type `ElementClass`. Inside a `SiteType`, connectivity of the site wires (and/or site nets) is described by a collection of `ElementPin` objects that each of a reference to their parent `ElementType`. Connectivity of a site wire is stored in each `ElementPin` and also in the `SiteType` object.

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

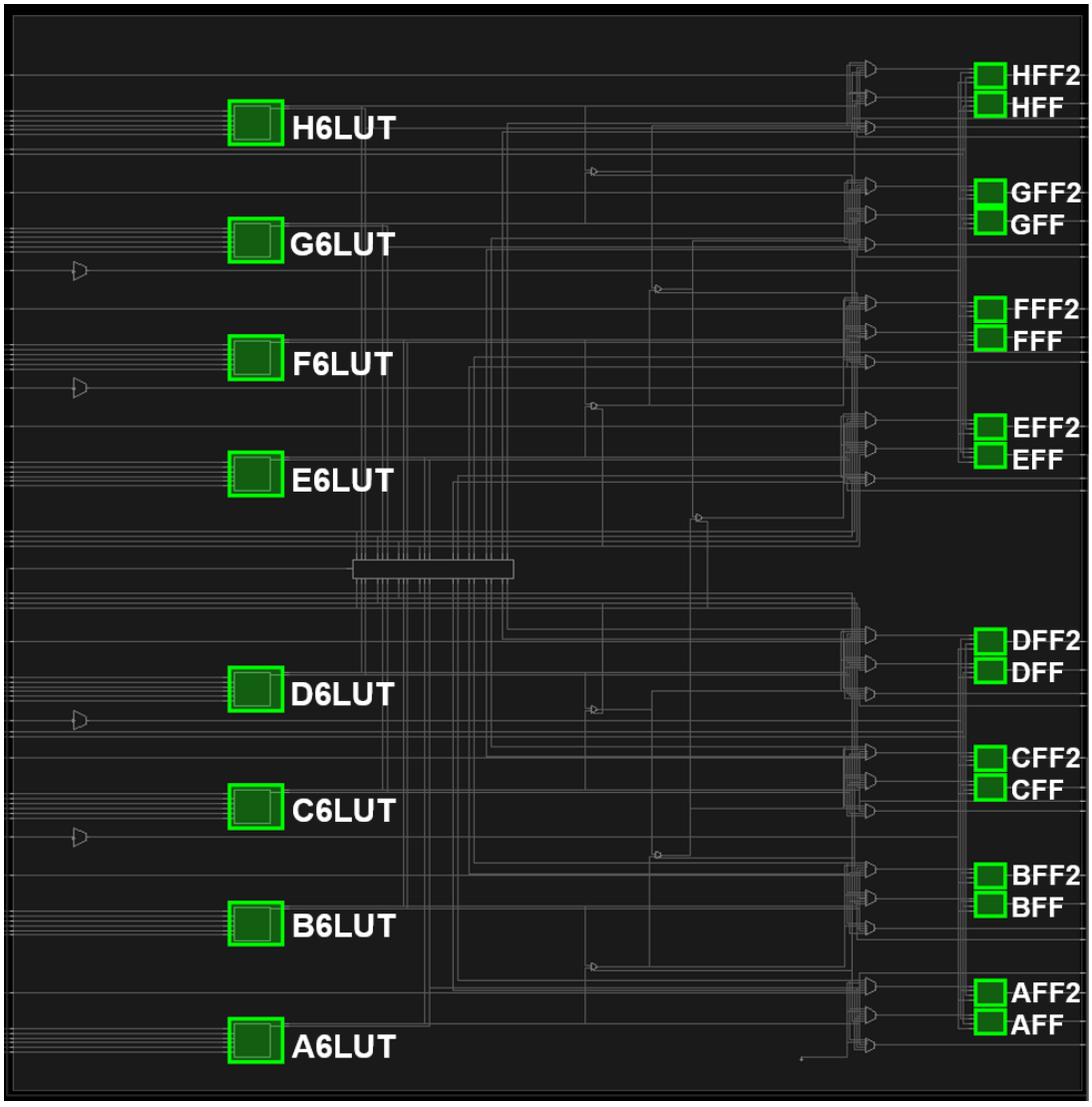


Fig. 6.2: A UltraScale SLICEL SiteType with the LUT and flip flop element names labeled

by providing faster lookups. In some cases, such as `TileTypeEnum` and `SiteTypeEnum`, the index has been explicitly enumerated and an enum is used instead.

Some objects, such as `Wire` and `Node` are generated on the fly as needed as there can be hundreds of millions of unique instances of each. Note that wires (as represented with an integer index) belong to a `TileType` and site wires (there is no explicit class for site wires, just an integer) belong to a `SiteType`. Thus, it is common to used raw `int` to store indices to wires and site wires and several APIs exist to aid in using them as such.

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 of choice for 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. Most of the classes (not exhaustively outlined here) follow closely to the keywords and constructs found in the EDIF file format. 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 explaination 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 `EDIFPortRef`. At one level, an `EDIFPort` belongs to an `EDIFCell` and an `EDIFPortRef` belongs to an `EDIFCellInstance`. Another distinction is that an `EDIFPort` can be a bussed-based object whereas an `EDIFPortRef` can only represent a single bit. An `EDIFNet` defines connectivity inside an `EDIFCell` by connecting `EDIFPortRef` objects together (port references on cell instances inside the cell or to external port references entering/leaving the cell).

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.

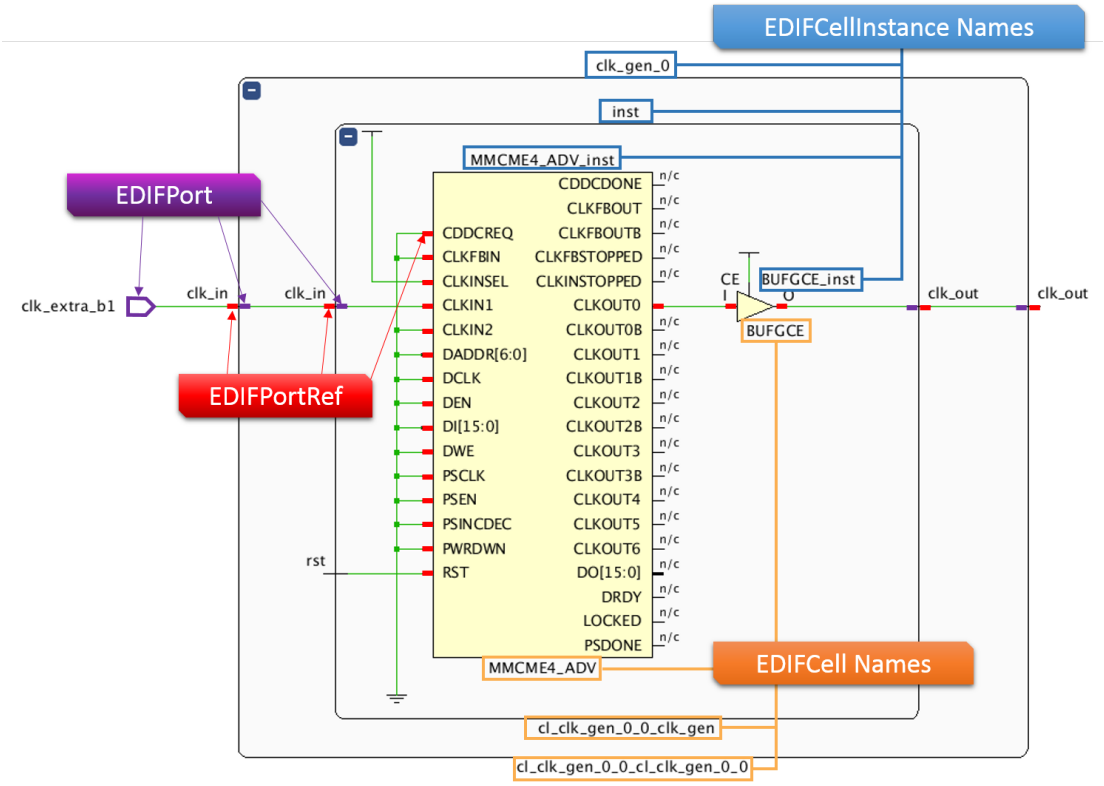


Fig. 6.3: Snapshot of the Vivado netlist viewer with references to RapidWright EDIF classes

6.3 Design Package (Physical Netlist)

The design package is the collection of all the physical components of a netlist's implementation. It contains all of the logical cell mappings to hardware, specifically the cells BEL/site 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, device and part references as well as other helper constructs. 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:

6.3.1 SiteInstance

Design representation and implementation in Vivado is element-centric (BEL and cell). This is in contrast to its predecessor, ISE and the XDL representation which made design representation more site-centric. The `SiteInstance` has less emphasis than in the past, but it cannot be abandoned. It keeps track of the cells placed onto its elements, the site PIPs used in routing and also a new construct called 'connection tag' that marks how routing resources map to nets.

Each `SiteInstance` maps to a specific compatible site within a device. The SiteInstance 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 element sites within the site. RapidWright also keeps track of a lock flag that is used in certain situations by Vivado to prevent components insides the site from being moved.

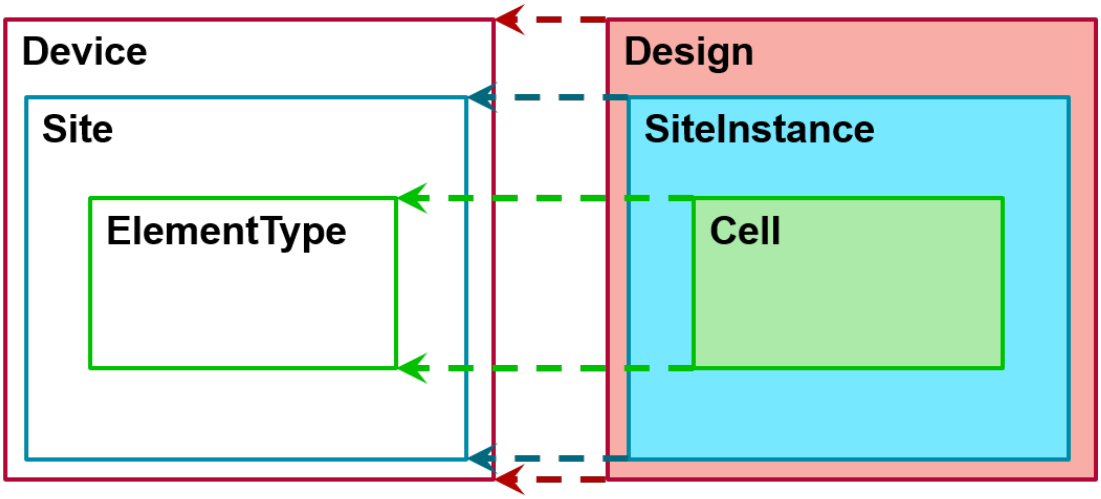


Fig. 6.4: Illustration representing how a `Cell`, `SiteInstance` and `Design` map to `ElementType`, `Site` and `Device` respectively

One of the more explicit parts of design representation that exist in Vivado over its predecessor is how it represents routing inside the site, or 'site routing'. Vivado handles routing inside of a site very differently than the routing outside of a site. Routing that takes place outside of a site is normally handled by Vivado's `route_design` implementation command. However, site routing is generally handles less explicitly and can be invoked anytime a change is made within a site. The two main components used to represent site routing are site PIPs (represented by the `SitePIP` class) and connection tags or CTAGs for short.

A site PIP is a programmable interconnect point that exists inside a site (specified inside a `SiteType`). Generally, a site PIP 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 SiteInstance is the entity 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 `SiteInstance` then it is interpreted as the site PIP being turned on or used.

A connection tag (or CTAG), is a mapping between a physical net name and a site wire name. As is commonly needed, a net being routed through a site will consume site wires and site PIPs as it traverses a path to its destination. CTAGs help make the path explicit by keeping a mapping of each consumed site wire to its corresponding net. Currently in RapidWright, this mapping must be accurately maintained by the user in order for designs to be properly represented when they reach Vivado. Failure to do so will result in unpredictable implementation behavior or errors when trying to load DCP files.

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. A `Net` is a physical net that implements a route using PIPs (programmable interconnect points) that, when combined together configure wires into a path from a source site pin to one or more sink site pins. The `PIP` class is instantiated as needed to save memory although there is a definition-like object called the `TilePIP` (device package) that contains definition data and belongs to the `TileType` class.

6.3.3 Cell

At the lowest level, a RapidWright cell maps a logical leaf cell from the EDIF netlist (`EDIFCellInstance`) to an `ElementType` (or logical BEL site). 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 phsyical cell pin mappings (pins on the `ElementType`).

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 `SiteInstance` 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 `ModuleInstance` 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 `ModuleInstance` class in the `design` package. Module instances have a unique name within the design and as each module has a collection of `SiteInstance` and `Net` objects, these containers are prefixed hierarchically with the module instance name. For example, if a module had a `SiteInstance` named "SLICE_X2Y2" and a `Net` named `data_ready`, a newly created module instance named "fred" would have counterpart `SiteInstance` 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 `ModuleInstance` concept is not available in Vivado or the DCP file format. If these constructs are used in a RapidWright design, they can be 'flattened' or saved to an XPN file (see next section).

6.3.6 Xilinx Physical Netlist (XPN) File Format

The Xilinx Physical Netlist (XPN) file format is a text-based file format that contains the placement and routing information of a RapidWright design. It contains the information most similar to an XDEF file in a design checkpoint, but it cannot be read by Vivado. The added benefit of the XPN file format is that it can also save and restore `Module` and `ModuleInstance` data along with all placement and routing information. There are four major sections in an XPN file, `PART`, `MACRO` , `SITE` and `NET`. The `MACRO` section is optional if the design does not contain any `Module` or `ModuleInstance` objects. Comments can be added by using the `#` symbol at the start of the line and the rest of the line will be ignored by the `XPNReader`.

```
PART <part name>
```

```
MACRO <name> <anchor site instance name> <implementation index> <min clock period␣
↪(ns)> [pblock]
        MACRO_INST <list of ModuleInstance objects instantiating this Module>
        MACRO_PORT  <name> <site instance name> <pin name> <worst case port delay␣
↪(ns)>
        ...
        SITE { see below for definition }
```

```
        ...
        END_SITES
        NET { see below for definition }
        ...
        END_NETS
END_MACROS
```

```
SITE <site name> <site instance name> <site type> [<LOCKED>] [MACRO_INST] [module␣
↪instance name] [template site instance name]
        BEL <bel name> <logical cell name> <library element name>
          PINS <bel pin name>:<library element pin name>[!] ...
        ...
        PIPS
          <element name> <input element pin name><pip arrow><output element pin name>
          ...
        CTAGS
          <site wire name> <net name>
          ...
...
END_SITES
```

```
NET <net name> [GND|VCC] [MACRO_INST] [module instance name]
  [OUTPIN <site instance name>.<site pin name>]
  [INPIN <site instance name>.<site pin name>]
  ...
...
  PIP <tile name>/<tile type name>.<wire0 name><pip arrow><wire1 name>
  ...
END_NETS
```

CHAPTER

SEVEN

DESIGN CHECKPOINTS

Table of Contents

- *Design Checkpoints*
  - *What is a Design Checkpoint?*
  - *What is Inside a Design Checkpoint?*
  - *Xilinx Design Language (XDL) vs. Design Checkpoint*
  - *RapidWright and Design Checkpoint Files*

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 actually a regular ZIP file that compresses several other files together into one. There are four main categories of files (Logical Netlist, Physical Netlist, Constraint, and Other) that are found in a DCP file as shown in the table below:

Category	Description	Name	Text/Binary	Encrypted?
Logical Netlist	EDIF Netlist	<design>.edf	Text	Encrypted if design secured
Logical Netlist	XN Netlist	<design>.xn	Binary	Encrypted
Physical Netlist	XDEF (Place & Route DB)	<design>.xdef	Binary	Unencrypted
Constraint	Early Constraints	<design>_early.xdc	Text	Unencrypted
Constraint	Constraints	<design>.xdc	Text	Unencrypted
Constraint	Late Constraints	<design>_late.xdc	Text	Unencrypted
Constraint	Board Constraints	<design>_board.xdc	Text	Unencrypted
Other	Metadata	dcp.xml	Text	Unencrypted
Other	Verilog Stub	<design>_stub.v	Text	Unencrypted
Other	VHDL Stub	<design>_stub.vhd	Text	Unencrypted

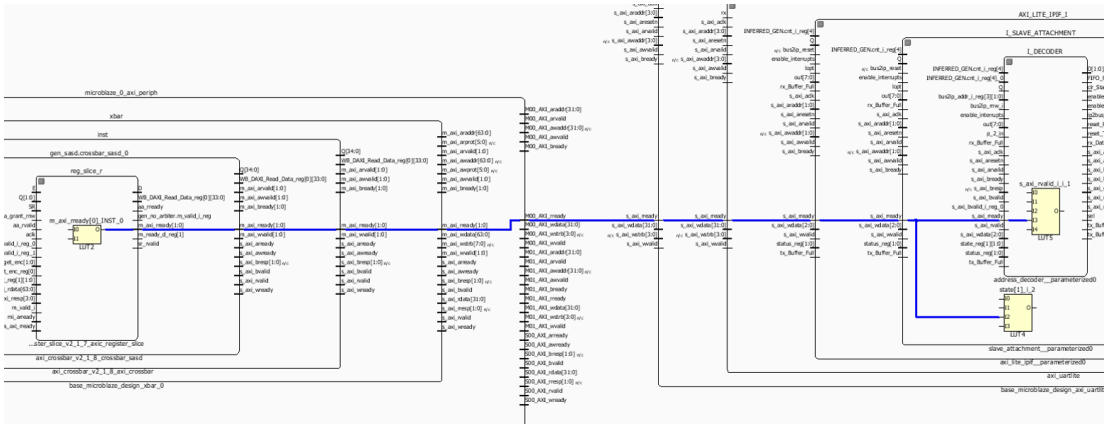
There are other files that get included within DCPs, but they will not be detailed here. Note that there are two copies of the logical netlist, one in a binary, encrypted file called XN and another in EDIF. As long as the design does not contain protected IP or has been secured, the EDIF file will be unencrypted and can be ready by RapidWright. RapidWright *cannot* decrypted encrypted files.

7.3 Xilinx Design Language (XDL) vs. Design Checkpoint

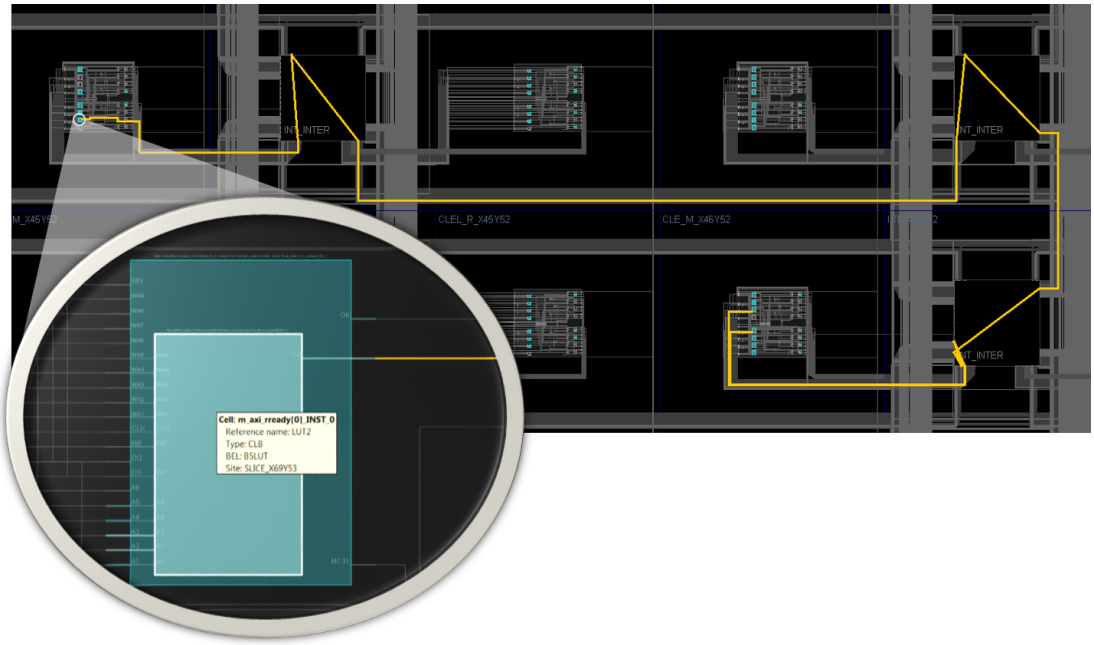
Prior to Vivado, users of Xilinx ISE could leverage tools like RapidSmith or Torc that relied on the Xilinx Design Language (XDL) in order to push and pull designs into an open source development environment. XDL is not supported in Vivado and thus obsoleted these tools for newer architectures. Despite some valiant efforts to bridge the gap in capabilities, there still remain long import runtimes for these alternative approaches.

When Xilinx created Vivado, it added some significant complexity to how implementation results were stored. These choices were made for good reason as it enables more complex netlist manipulation during the implementation flow that was quite difficult previously. However, XDL was no longer a sufficient vehicle to carry the necessary information. For example, the XDEF file found within the design checkpoint is the closest representation to the information content of XDL. However, this file alone is no longer sufficient to continue implementation processes or generate bitstreams.

The addition of the full hierarchical netlist and user constraint files in the design checkpoint are the main differentiators between XDL and design checkpoints. XDL only had a single level, physical netlist of placed and routed cells. Design checkpoints, in contrast, maintain the full hierarchy of the user's original intent from RTL. To illustrate the added complexity, consider the diagram below:



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. In Vivado's design checkpoints, any modification to the netlist must be managed in the hierarchy, however, in XDL, there would have been only one net represented, the physical net as shown in the diagram below:



Overall, design checkpoints increase the complexity of handling designs, but they also open up more possibilities to innovate than with XDL.

7.4 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 name (except with the .edf extension).
- 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");
```

Most of the classes and code relevant to dealing with design checkpoint files are in the `com.xilinx.rapidwright.dcp` package.

CHAPTER

EIGHT

IMPLEMENTATION BASICS

Table of Contents

- *Implementation Basics*
  - *Placement*
  - *Routing*

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/element 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 the BEL to a `Cell` object 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 `SiteInstance`)
4. Addition of CTAGs also as part of routing a site (stored in the respective `SiteInstance`)

Generic pin mappings are available in the `UnisimManager` by calling `getPinMappings()` although the pin mappings should be reviewed/updated to match the respective placement scenario.

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 SiteInstance 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.

A CTAG (as you may recall from *SiteInstance*) is a mapping between a site wire and the respective net that is being routed on that resource. The mappings are essential and can be tricky to get correct as there are a lot of different scenarios where they may be required. RapidWright is still maturing and currently has limited facilities to answer all these questions automatically. Work will continue to make this process easier and more reliable. However, for the most reliable placement of a cell, an authoritative source is to attempt a single cell placement within Vivado in the same context using the `place_cell` Tcl command. Compare the state of the site before and after the placement to get a definitive recipe.

8.1.2 Site Placement

Within RapidWright, it is quite straightforward in most cases to move a `SiteInstance` 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 light overview of each.

8.2.1 Site (Intra-site) Routing

When a cell or cells are placed onto BELs, typical Vivado convention is to route the site immediately after. Routing a site implied mapping the physical net to site wires and site pips. This is done within the respective `SiteInstance`

by using CTAGs to map nets to site wires and turning on SitePIPs. Site routing is quite tricky and often it might be best to observe examples within Vivado itself to get CTAG and SitePIP configuration correct.

8.2.2 Interconnect (Inter-site) Routing

The bulk of the work load in routing a design is in inter-site routing. This is the task of routing site pin to site pin using the PIPs available within the general interconnect found in switch boxes (INT tiles). 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. . .)

8.2.4 Route Analysis

One of the most useful reports for routing generated by Vivado is created by running `report_route_status`. It has several options and can be configured to dump explicit node by node connectivity. In this format there are symbols generated on each line, here is a table that defines their meaning:

Symbol	Description
`[`	Tree End and Branch Start
`{`	Branch Start
`*`	Locked Node
`!`	Inverting Arc
`#`	Arc Set to Invert
`}`	Branch End
`]`	Tree End and Branch End
`p`	

There are various types of nodes that do not correspond to those in the device but are used for structure of the route. The names that appear in reports generated from `report_route_status` are shown in the table below:

Node Type (prefixed by "NoTile/")	Description
`Logical_Driver`	Designates the source site for a net when ambiguous
`GAP`	Makes a gap in routing valid
`GAP (PlaceHolder: <Tile/Wire>)`	Place holder with info. to restore clock routing
`NoWire`	An invalid node
`Start_of_Stub`	Validates an antenna node
`Delay_Arc_Index=<Delay Arc Index>`	Captures variable delay settings in routing

CHAPTER

NINE

XDD FILE FORMAT

This section describes the XDD (Xilinx Device Description) file format used by RapidWright. An XDD file provides an exhaustive description of a particular Xilinx device supported by Vivado and can be generated using the `XDDPrinter` class.

An XDD file contains seven different sections that describe a device's resources and connectivity. Due to the highly regular nature of Xilinx devices, there are many constructs that are replicated. In a loose sense, there are two kinds on constructs in the XDD files: definitions and instances. Definitions are abstract models of a construct intended to be instantiated somewhere on the device one or more times. The following sections represent lists of unique definitions: Tile Patterns, Node Templates, Tile Types, and Site Types. Tiles and Clock Regions are instance constructs and are populated by instances of the definition constructs. For example, a Tile is an instance of a Tile Pattern and a Tile Type. A Tile may also have Sites (instances of Site Types) within it.

The Intent Codes section is a named enumeration for annotation of wires.

Under the heading of each section below, a boxed textual construct pattern is shown to aid the reader correlate what kinds of information can be found in the XDD file. These textual construct patterns can also be found in the comments (any line beginning with a '#') above each section in the XDD file. A common pattern used is providing the total count of 'child' objects of each section to aid in parsing. Additionally, most every construct is enumerated sequentially with the intention that such data could be stored easily in an array. By providing universally consistent enumerations for all objects (wire indices within a tile, for example, refer to the same named wire in all contexts), data structures created to represent these objects can use integer comparison instead of string comparison, potentially saving the user runtime and memory.

Note to the reader: The subsections presented below are in the order in which they appear in the XDD file. This may not be the most logical order for understanding the constructs, the user is encouraged to jump to the subsections of interest as questions arise.

9.1 Tile Patterns

```
(tile_patterns <tile_pattern count>
  (tile_pattern <tile_pattern ID> <tile_type name> <template_entry count>
        (template_entry <wire ID> <wire name> <node template ID> <template offset>)
        ...
  )
  ...
)
```

Every tile has a tile pattern that provides a mapping (template entry) for each wire to a node template (see *Node Templates* section below). To illustrate this concept, consider the figure below where five node templates are illustrated in a set of adjoining tile types (note these are tile types, not actual tiles).

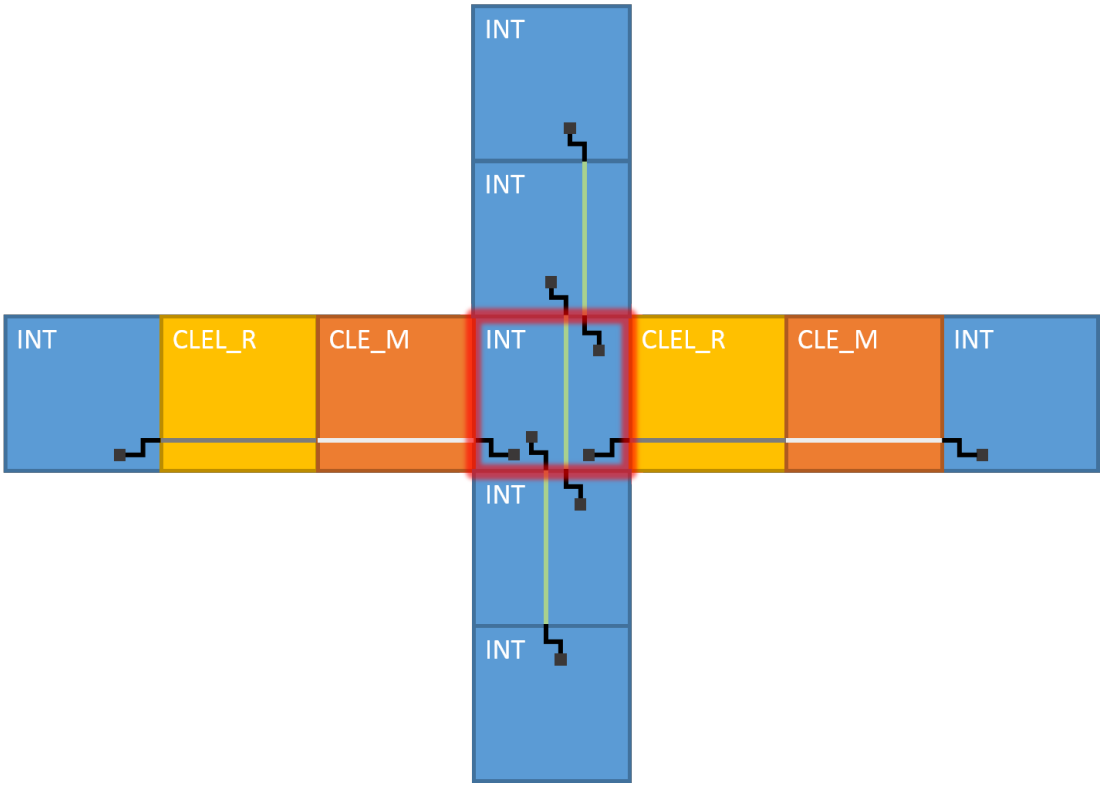


Fig. 9.1: Simple illustration of file node templates in an INT tile.

There are five wires that belong to the INT tile type highlighted in red. Each of these wires belongs to a node template. A node template is characterized by a list of wires that belong to different tile types and their positional offset to a base wire or origin wire. If two tiles use the same tile pattern, they both use the same set of node templates. In Vivado, a tile pattern index is present on each tile as the property `TILE_PATTERN_IDX`. For example, a set of 8 tiles that have the same pattern are highlighted in magenta in the figure below.

Each tile pattern listed in the XDD has a tile type and a list of node template entries. Each template entry maps each wire ID/name within the tile to its corresponding unique node template ID to which the template in this tile pattern corresponds (see *Node Templates* for more details). In tiles that have wires that do not connect to sites or PIPs but simply pass over the tile, they are called flyover wires. Flyover wires do not have their template entries listed for space purposes as it is partially redundant.

9.2 Node Templates

```
(node_templates <node_template count>
  (node_template <node_template ID> <wire_item count>
        (wire_item <template offset> <delta X> <delta Y> <tile type>.<wire name>
↪<wire ID>)
        ...
  )
  ...
)
```

This section of the XDD file enumerates all unique node templates in a device.

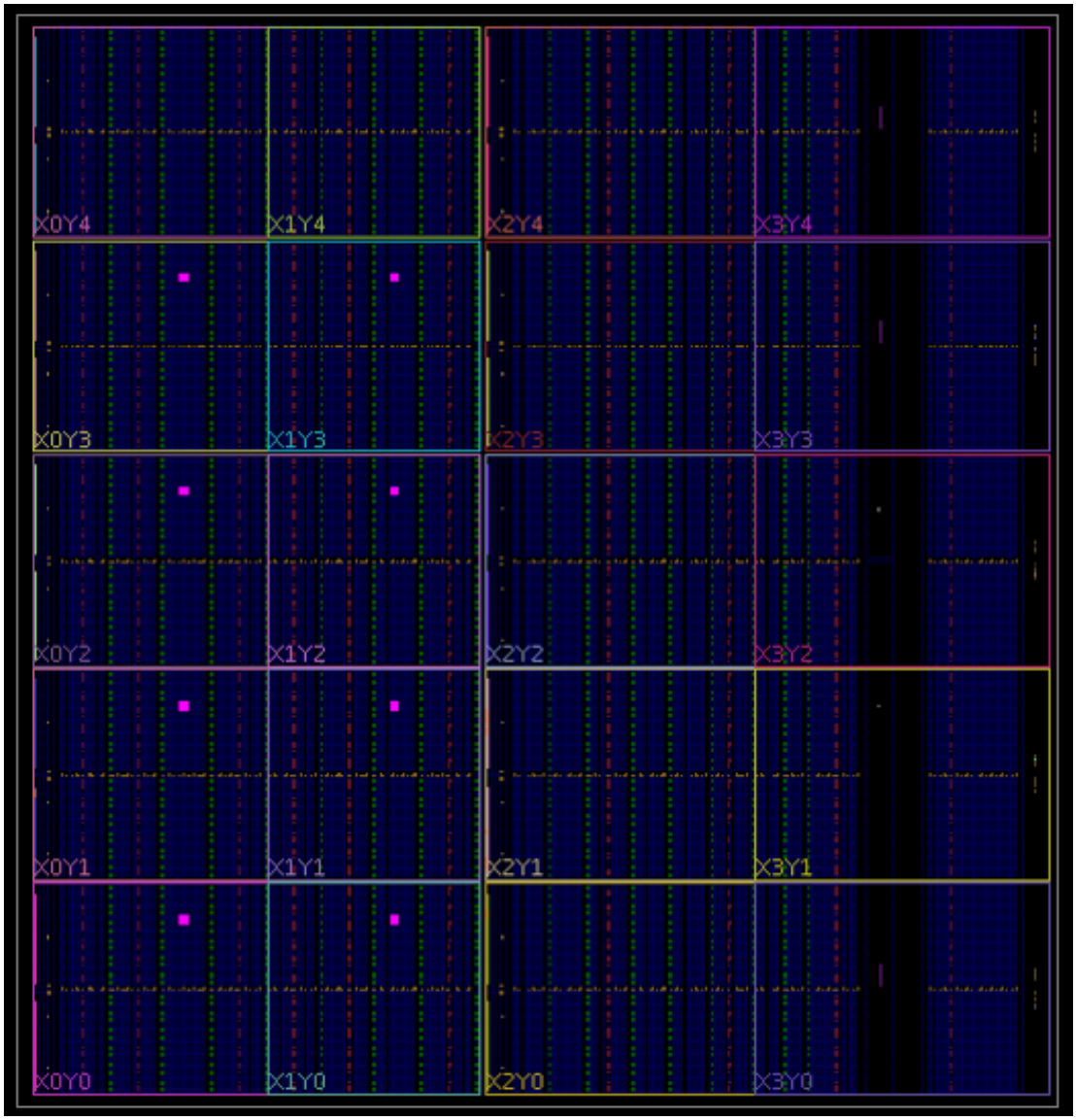


Fig. 9.2: Tiles with the same pattern highlighted in magenta.

9.2.1 Node (Review)

Before describing node templates, let us first review the concept of a node. Each tile type (see *Tile Types*) has a set of named, enumerated wires defined within its boundaries. As tiles are instantiated to build a contiguous fabric, by design, the wires line up with and abut to wires in neighboring tiles. For example, consider the figure below where each tile type shows a wire (most tiles have several wires, only one per tile is shown here for simplicity) within its boundaries that forms a larger wire when the two tiles are abutted together.

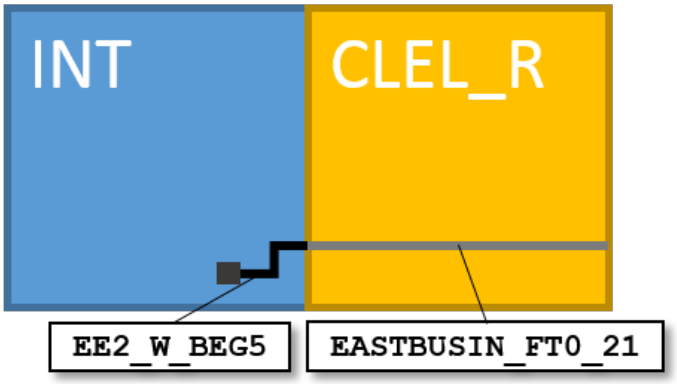


Fig. 9.3: Two tiles abut connecting the wires EE2_W_BEG5 and EASTBUSIN_FT0_21

The complete set of electrically equivalent connected wires is known as a node. The figure above only shows a portion of one node, a complete node can be seen in the figure below.

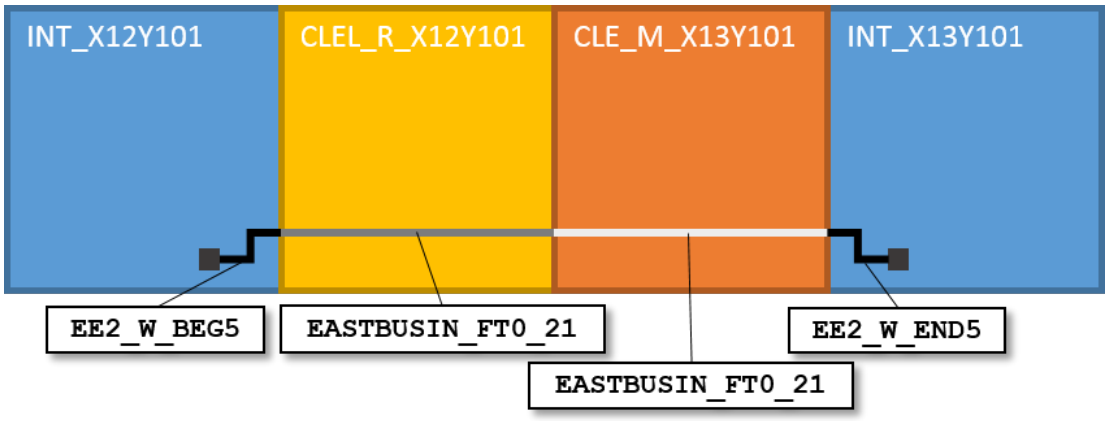


Fig. 9.4: Showing the node INT_X12Y101/EE2_W_BEG5 spanning 4 tiles

A wire is only defined within the context of its parent tile type, but to reference a node, the tile instance must be known as it includes all of the touching wire segments from abutting tiles as shown in Figure 11. Nodes within Vivado are called by their source tile and source wire, so the node illustrated in Figure 11 is called INT_X12Y101/EE2_W_BEG5. In summary, a node is defined as a collection of one or more wires that are electrically equivalent.

9.2.2 Node Template

The concept of a node template is used to help reduce the volume of data that would be required to represent every instance of a node on a device. Several nodes follow patterns based on the tile types and wires involved due to the repetitive nature of the fabric. Node templates are the prototype or definition of a canonical node that has one or

more instances on the device. For example, visual illustrations of two different node templates are shown in the fogire below. Note that there is only a single difference between (a) and (b), which is the inclusion of an additional tile type, CFRM_CBRK_L. Even though these two nodes may present the same connectivity, the nodes are indeed different and belong to different node templates.

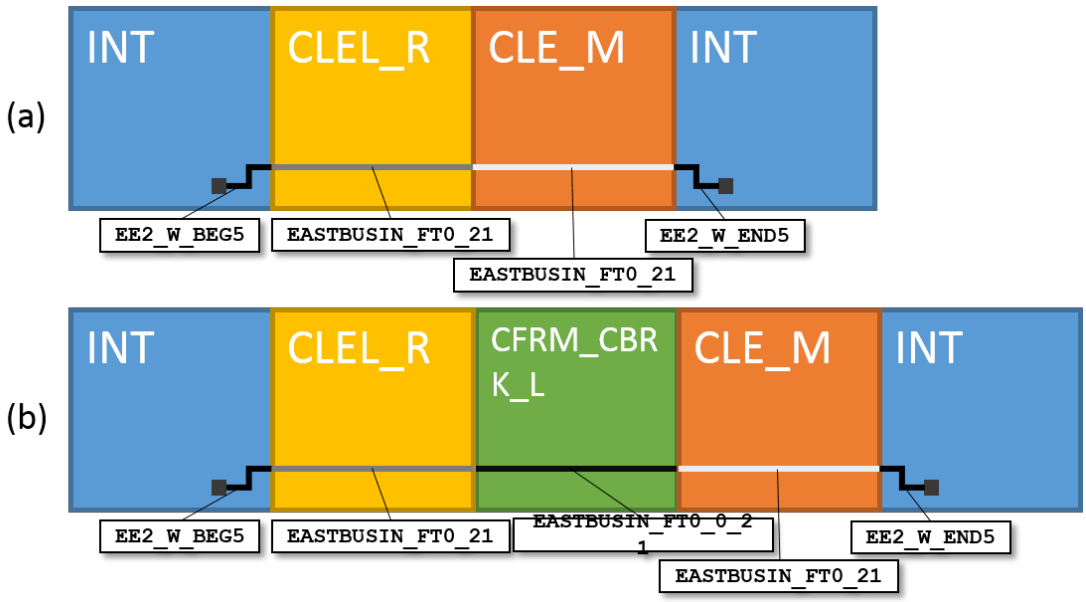


Fig. 9.5: Node templates for the same architectural wire in two different contexts

9.2.3 Wire Item

In order to describe a node template, the notion of a wire item is used. A wire item describes the type of wire (tile type and wire name) and its relative offset from the origin wire in the node template. The origin wire is generally defined as the source wire or the closest wire to the source of a potential signal. For bidirectional wires, this convention is less obvious. As an example, the node template illustrated in part (a) of the figure above would have the following text description:

```
(node_template 197235 4
        (wire_item 0 0 0 INT.EE2_W_BEG5 304)
        (wire_item 1 3 0 INT.EE2_W_END5 815)
        (wire_item 2 1 0 CLEL_R.EASTBUSIN_FT0_21 126)
        (wire_item 3 2 0 CLE_M.EASTBUSIN_FT0_21 128)
)
```

There are four relative wires within the node template. Note that the order of the wire items is not contiguous relative to their connectivity. The wires with more PIPs on them appear first (which if stored in memory this way is useful during routing expansion).

9.3 Intent Codes

```
(intent_codes <intent code count> <intent type>
   (intent_code <intent code> <intent code name>)
        ...
)
```

An intent code is a property applied to wires and nodes. The intent code of a wire or node can be reported in Vivado using the `report_property` command of the object. Additionally, the intent code name is displayed in the Node Properties box in the Vivado GUI.

This section of the XDD file maps the integer intent code to its name. Intent codes can be used to help classify different wires and nodes to their respective types to aid in algorithms such as routing to help prune expansion selection. For example, consider the sparse clock resource representation in the figure below. Each of the clocking resource types have a specific intent code that is color coded. These intent codes are useful in clock routing as different types of wires are provided to route and distribute a clock signal from its source to all the sinks. By having a quick identifying classification, a clock router can quickly discard any wires or nodes that do not fit the kind of resource needed during each stage of clock routing and distribution.

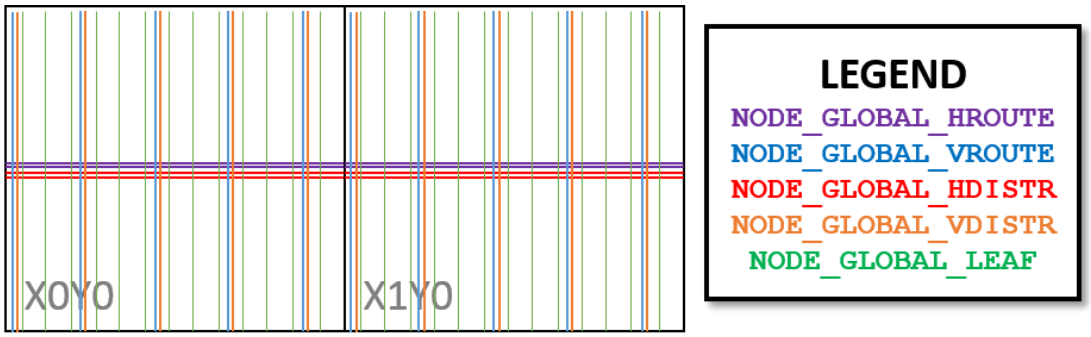


Fig. 9.6: Clock routing resources within two clock regions with color coded intent codes

9.4 Site Types

```
(site_types <site_types count>
  (site_type <site_type ID> <site_type name> <sitepin count> <sitewire count>
↪<element count> <siteconn count> <sitepip count> <reserved> <checksum> <primary_
↪type>
         (secondary_site_types [site_type]+ )
         (sitepin <sitepin ID> <sitepin name> <sitepin direction> <RESERVED>)
         ...
         (sitewire <sitewire ID> <sitewire name>)
         ...
         (element <element ID> <element name> <element def type> <element type>
↪<elementpin count>
           (elementpin <elementpin ID> <elementpin name> <direction> <sitewire name>)
           ...
         )
         ...
         (siteconn <siteconn ID> <elementpin name> -> <elementpin name> <sitewire␣
↪name>)
         ...
         (sitepip <sitepip ID> <element name>.<elementpin name><pip arrow><elementpin␣
↪name>)
  )
  ...
)
```

The site types section of the XDD file enumerates all site types in a given device family. A site type is the prototype

definition of a site. A common example of a site type is a SLICEL or SLICEM. Site types can be primary (there are physical sites of the same type in the architecture) or secondary (the type can be placed on sites of the secondary's primary type, but generally no secondary sites of this type exist). Secondary site types represent additional functionality where a single site type could not fully represent all of the hardware's capabilities. Note that secondary site types will have a pin mapping back to the primary site type pins as those are the names associated with wires in the routing of the device.

Site types are composed of several constructs: site pins, site wires, elements, element pin connections and site PIPs. A site type will define an interface of input and output pins, called site pins. All user signals enter and exit a site using a site pin. A site wire is a collection of electrically equivalent element pins. For example, consider the highlighted site wire SLICE_X10Y164/AX in the figure below. It connects five element pins: AX (corresponding site pin), CARRY8/AX, FFMUXA1/BYP, A6LUT/DI2 and F7MUX_AB/S0.

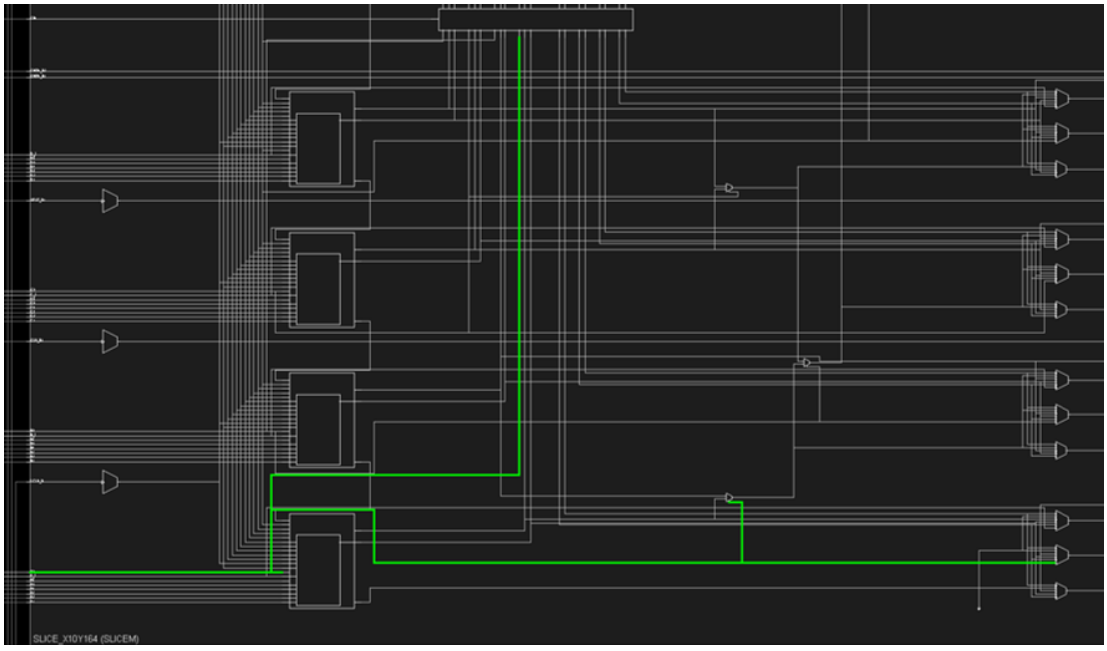


Fig. 9.7: Highlighted site wire AX in SLICE_X10Y164 showing 5 element pin connections

9.4.1 Elements

Elements are the lowest level functional blocks present on the device. Elements belong to a parent site type. Elements within the same site type will have a unique name and will have a corresponding definition type (i.e. some elements are instances of the same library element). There are three classes or types of elements: BELs, RBELs and ports.

BELs (basic elements of logic) consist of LUTs, carry chains, flip flops and similar kinds of programmable elements. BELs can also share element pins, such as in the 6LUT where obeying certain rules can also support a 5LUT cell at the same time. BELs can also support route through behavior when a logic cell is not consuming its resources.

RBELs (routing BELs) only support routing, with the single exception of providing an inversion option in key locations. RBELs are programmable muxes within a site to help connect site wires together. The programmable connections provided by an RBEL from its input to output pins are called site PIPs. Port elements are simply the internal side of a site pin. They provide an element pin internal to the site to help element pins and site wire abstractions to work together consistently. The name of the port element and the site pin are one and the same.

The site connections construct, siteconns, describes element pin to element pin connections within the site. This list of connections define all the element pins that belong to a site wire.

9.5 Tile Types

```
(tile_types <tile_types count>
  (tile_type <tile_type ID> <tile_type name> <site count> <wire count> <pip count>
  ...
        (site_type_inst <site_type_inst index> <site_type ID> <site_type name>)
        ...
        (wire <wire ID> <wire name> <intent code name> <RESERVED>)
        ...
        (pip <pip ID> <tile_type>.<wire0 name><pip arrow><wire1 name> <RESERVED>
↪<RESERVED> <is pseudo> <is test> <is excluded> <is inverted>)
        ...
  )
  ...
)
```

The tile types section of the XDD file lists all of the unique tile types used throughout a device family. A tile type consists of three potential groups of objects: (1) site type instances, (2) wires, and (3) PIPs.

9.5.1 Site Type Instances

A tile type may have one or more site types instantiated, each as a child object. These site type instances are simply instances of site types (as defined in the Site Types section). As tile types serve as the prototype definition of a tile, it must have a notion of the kinds of sites tiles will contain. In the Tiles section, the actual site presented to the user of the fabric is defined and is given a unique instance name.

9.5.2 Wires

Each tile type has an enumerated ordered set of wires with unique names. These wires exist outside of sites but inside tiles. They connect the edges of tiles to other constructs within tiles such as switch box/PIP junctions (in interconnect tiles) and site pins.

9.5.3 PIPs

A PIP (programmable interconnect point) defines a potential connection that can be made between two wires (and in a fabric context, two nodes) using a routing mux (not explicitly represented). A route between two site pins is generally established by turning on a select set of PIPs which connect the wires in between the two site pins together.

A PIP exists outside of sites but inside tiles (PIPs that exist inside sites are called Site PIPs and are described in the Site Types section). Generally, PIPs exist within a switch box, however, this construct is not present in the VRI, only in the Vivado Device Model GUI view. A PIP always connects two wires that are within its own tile. Note however, that in a fabric context when tile types are instantiated in a grid, wires at the tile edges can form connections by abutment. The device model presented in the VRI does not include explicit tile edge connections. The VRI provides a wire-centric view and the user should be aware of the tile-based fabric construction.

PIPs have different properties and types. To denote a class or type of PIP, the ASCII PIP arrow in between wire0 and wire1 of the PIP is used. The different variants of the PIP arrow are shown in the Table below.

ASCII Arrow	PIP Type
->	Directional, not buffered
->>	Directional, buffered
<->	Bidirectional, not buffered
<<->	Bidirectional, buffered
<<->>	Bidirectional, buffered

9.6 Tiles

```
(tiles <tile rows count> < tile columns count>
  (tile <row index> <column index> <tile name> <tile_type name> <tile_pattern ID>
↪<site count>
        (site <site ID> <site name> <site_type name> <is internal> <RPM X> <RPM Y>
↪<pinwire count>
          (pinwire <pinwire ID> <pinwire name> <direction> <wire name> <node tile␣
↪name> <node wire name>)
          ...
        )
        ...
  )
  ...
)
```

A tile is an instantiation of a tile type. It is made unique by giving it absolute tile coordinates, a unique name and the sites within the tile are also given unique names. Vivado relies on the absolute grid coordinates of the tiles and sites to establish relative proximity during algorithmic execution. Tiles are also assigned a tile pattern (see `Tile Patterns`).

As sites will connect to wires and in turn nodes, the name of the node that each site pin connects to is also listed. This is useful in applications such as routing where the name of the node is the input pin within the switch box (INT tile) neighboring the site. The node is a better indicator of how a route should enter the site.

9.7 Clock Regions

```
(clock_regions <clock_region rows count> <clock_region columns count>
  (clock_region <row index> <column index> <clock_region name> <upper right tile name␣
↪start>:<lower right tile name end>)
  ...
)
```

Clock regions (also known as fabric sub regions, or FSRs) are groups of tiles that share clocking resources. All tiles in a clock region can be clocked from the same clock distribution lines.

For brevity, a clock region is described by identifying the upper left tile and lower right tile that form an inclusive rectangle around the tiles that belong to a clock region. Clock regions are named after their XY coordinates within the fabric grid with the origin being at the lower left corner of the fabric. It is interesting to note, that not all tiles fall into a clock region. There are some exceptional tiles at clock region boundaries that do not belong to a clock region.

CHAPTER

TEN

A PRE-IMPLEMENTED MODULE FLOW

This section presents 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.

10.1 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.

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`.

This command runs on a currently open IP Integrator design in Vivado, builds all IPs out-of-context, invokes the BlockStitcher in RapidWright to stitch all of the pre-implemented blocks together, places the blocks and routes them into a final implementation. 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.

10.1.1 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 ('.impl.guide') directing the flow to create multiple unique implementations of the same module/IP.

10.1.2 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.

10.2 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.

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.

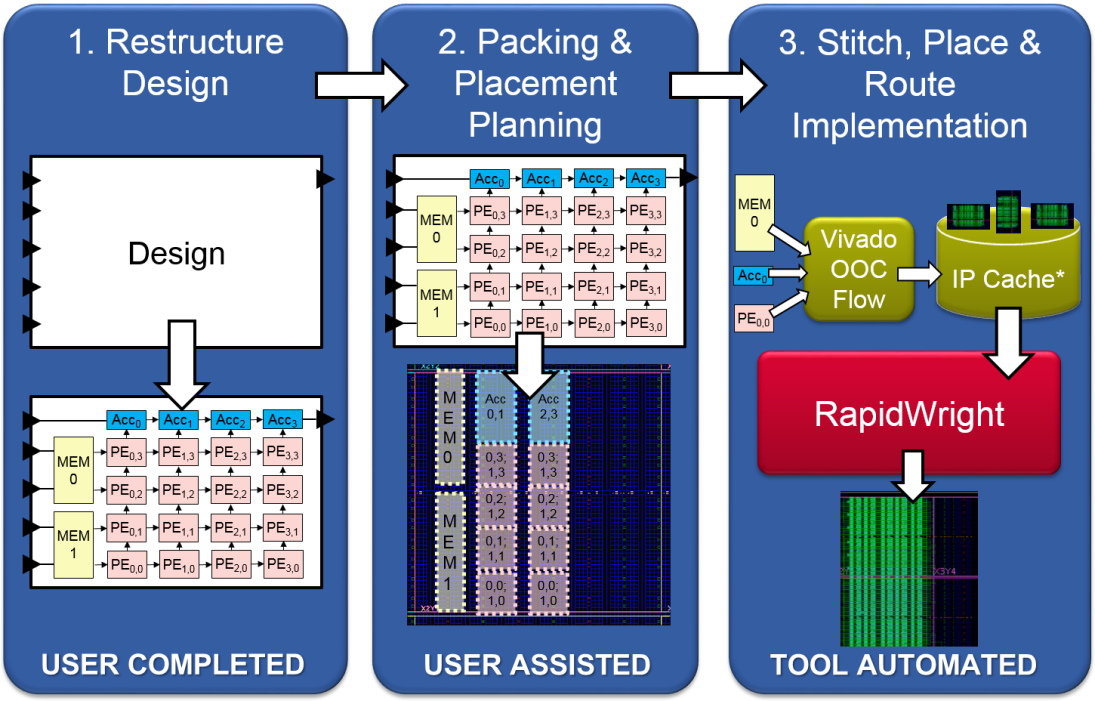


Fig. 10.1: High level visual of the three step process for the high performance module-based design strategy

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 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.

10.2.1 Implementation Guide File

An implementation guide file 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 ...

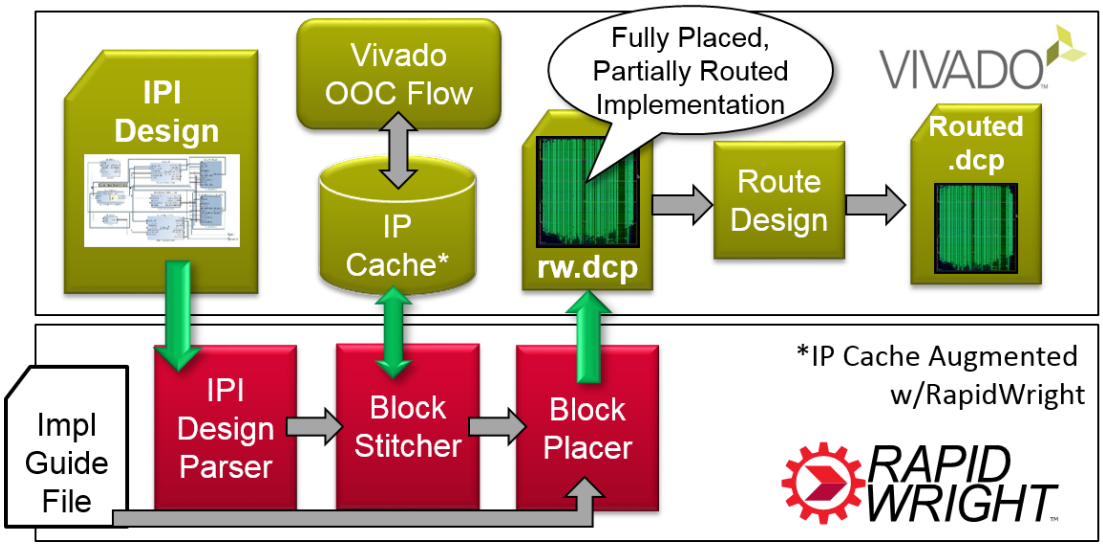


Fig. 10.2: High level view of the pre-implemented flow process and interactions between Vivado and RapidWright

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
```

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.

10.3 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.

10.3.1 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.

10.3.2 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.

10.3.3 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.

CHAPTER

ELEVEN

RAPIDWRIGHT TUTORIALS

11.1 Insert an ILA (ChipScope) into a Routed DCP

What You'll Need to Get Started:

- Vivado 2017.3
- RapidWright 2017.3

Often, you might find yourself in the following scenario. You've worked tirelessly to finally meet timing on your design and you are full of anticipation while you patiently generate a bitstream. Your anticipation turns to anguish and horror as you realize your design is not working in hardware. You fret over what unpleasant and demeaning tasks you'll have to perform just to figure out what is going wrong. All the while you are realizing that all your hard work in closing timing has gone down the drain. Unless. . . , perhaps, if there was some way you could keep your design intact and insert an ILA (ChipScope) core on top of your design without disturbing the placement and routing. This tutorial will help you do just that.

Since this is a tutorial, we will start by showing you how to generate a placed and routed design from scratch to test. If you have your own, know that it will need to meet the following requirements:

- It cannot already have a Debug Hub present in the design (this is common if you are using a MIG or other Debug core)
- There must be adequate resources for the ILA you are requesting present on the device

To generate an example placed and routed MicroBlaze design, run the following Tcl commands in your Vivado's Tcl prompt (Note the use of $CWD, please replace this with your current working directory):

```
create_project microblaze $CWD/microblaze -part xcku040-ffva1156-2-e
set_property board_part xilinx.com:kcu105:part0:1.2 [current_project]
set_property target_language VHDL [current_project]
create_bd_design "base_mb" -mode batch
update_compile_order -fileset sim_1
update_compile_order -fileset sources_1
launch_runs impl_1
```

This will take several minutes. When this is finished, we need to generate unencrypted EDIF (the EDIF in the DCP will by encrypted by default) by running the following commands:

```
open_run impl_1
write_edif $CWD/microblaze/microblaze.runs/impl_1/base_mb_wrapper_routed.edf
```

Once you have the DCP and EDIF files ready, we can insert the ILA:

let's run the ILAInserter.java class in RapidWright by running:

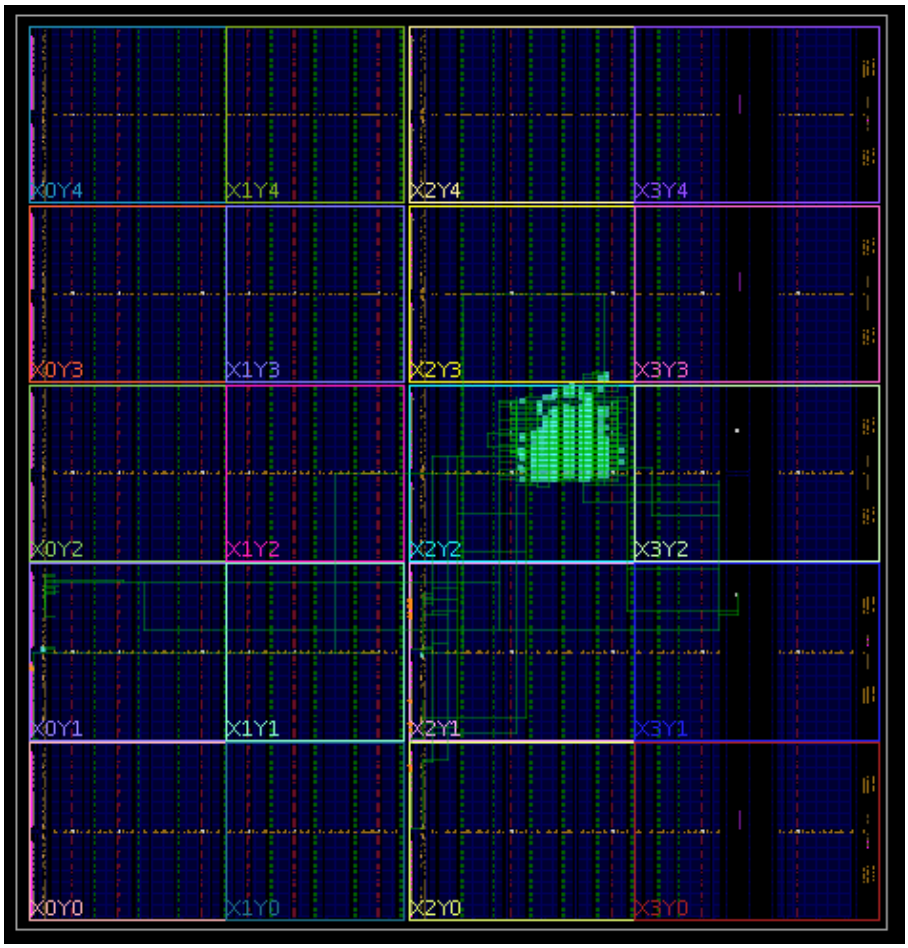


Fig. 11.1: The implemented microblaze design should looks similar to the one above

```
java com.xilinx.rapidwright.debug.ILAInserter base_mb_wrapper_routed.dcp
microblaze_with_ila.dcp 16 1024
```

The 16 specifies the probe count of the ILA to be generated and 1024 is for the capture depth. This script will automatically synthesize an ILA for you and stitch it into the DCP with these parameters. This will again take several minutes, but when it completes successfully, you'll have your placed and routed design with an ILA inserted.

Currently, this tutorial requires you to place and route the ILA on top of the existing placed and routed design. To finalize the implementation run the following commands:

```
open_checkpoint microblaze_with_ila.dcp
place_design
update_clock_routing
route_design
```

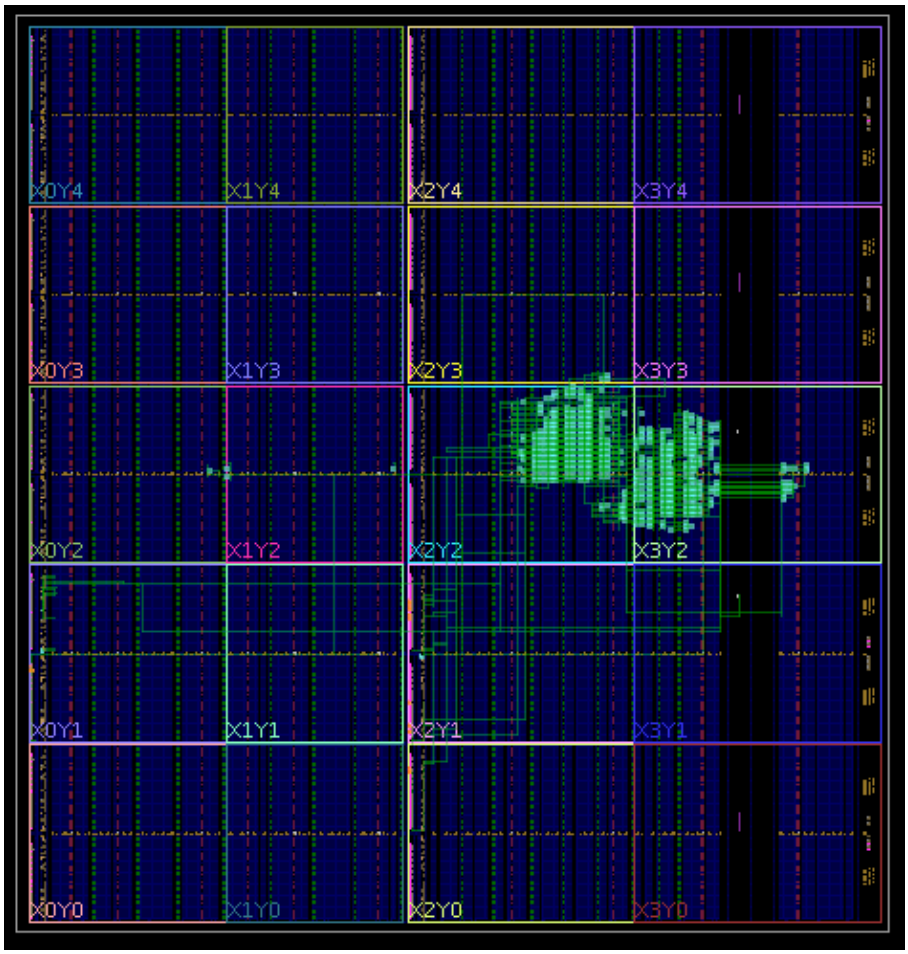


Fig. 11.2: After placing and routing the ILA on top of the original design, your implementation should look similar to that above

Now, using Vivado's ECO cockpit (Layout->ECO), you should be able to route the probes to any signals of interest.

11.2 Build an IP Integrator Design with Pre-Implemented Blocks

What You'll Need to Get Started:

- Vivado 2017.3
- RapidWright 2017.3

This tutorial will begin 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

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:part0:1.3 [current_project]
set_property target_language VHDL [current_project]
create_bd_design "base_mb" -mode batch
instantiate_example_design -design base_mb base_mb
```

Now that we have an IPI design created, we have the option of running one of two different implementation flows. To run the conventional implementation flow in Vivado, you could run:

```
launch_runs impl_1
```

But this tutorial is here to show you how to build this design with pre-implemented blocks. To run the RapidWright pre-implemented block flow, you will need to source a rapidwright.tcl and then run a command:

```
source ${::env(RAPIDWRIGHT_PATH)}/tclScripts/rapidwright.tcl
rapid_compile_ipi
```

This Tcl procedure will invoke Vivado to generate and synthesize the various IPs 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 placed and routed DCP.

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.

11.3 Create Placed and Routed DCP to Cross SLR

What You'll Need to Get Started:

- RapidWright 2017.4 or later
- Vivado 2017.3 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.

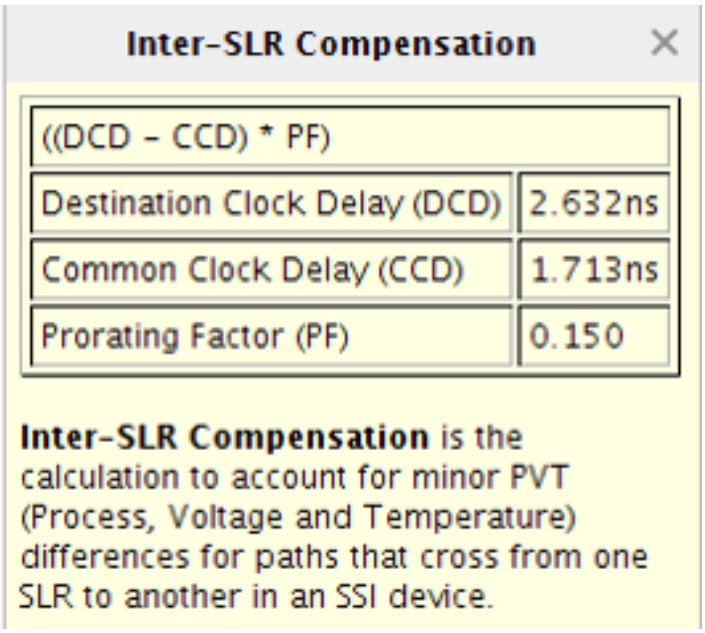


Fig. 11.3: Example Vivado tooltip window describing the Inter-SLR Compensation delay penalty

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.

11.3.1 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:
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 2017.3.0.

// 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);
```

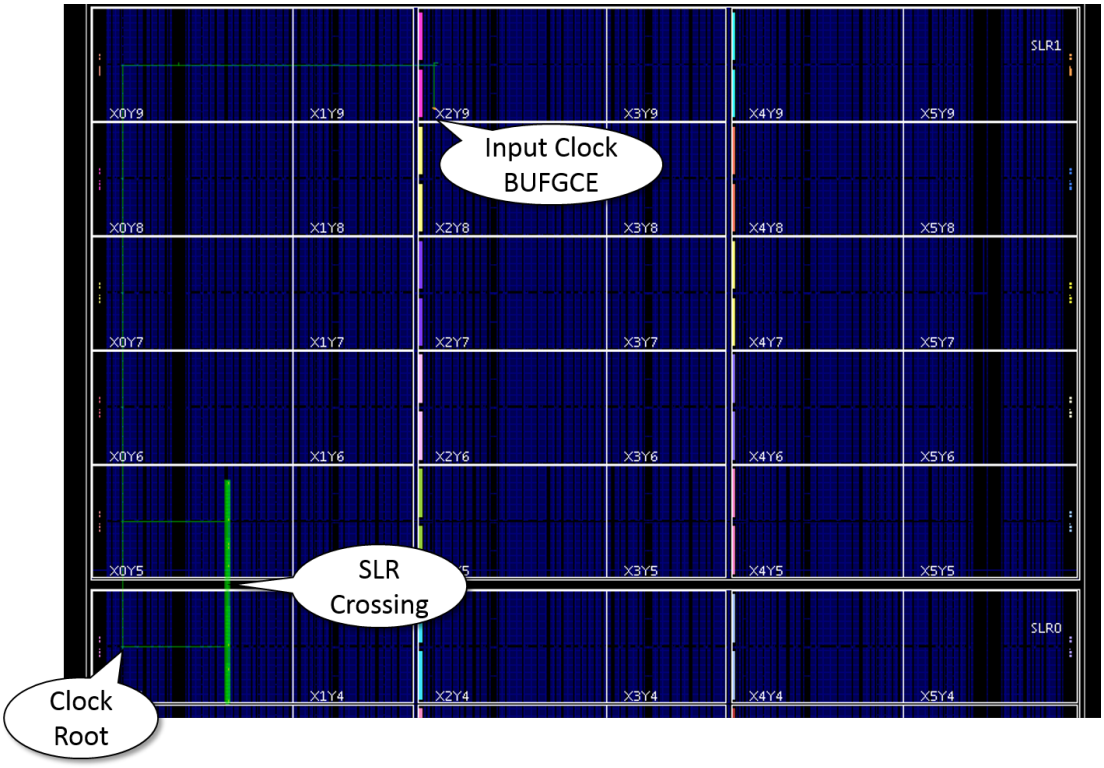


Fig. 11.4: Vivado Screenshot with bubble annotations of a single, bi-direction 512-bit SLR crossing circuit.

```
  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:

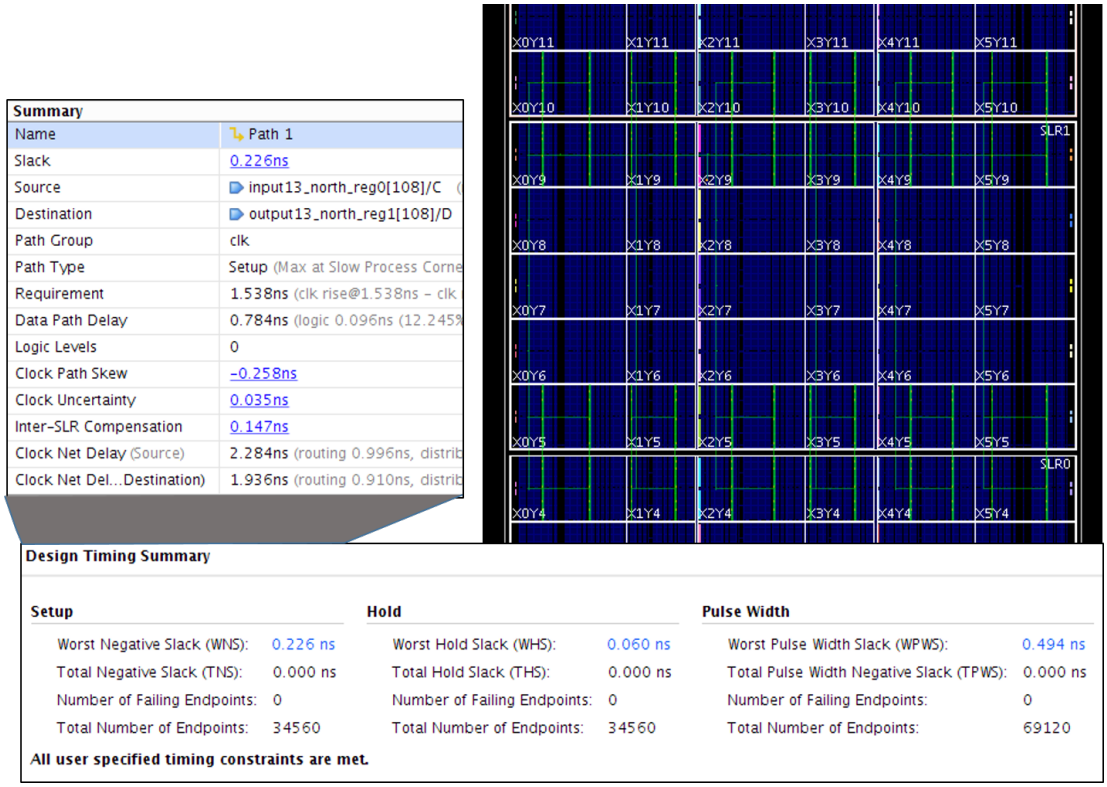


Fig. 11.5: Vivado Screenshot of all SLLs being used at potentially a 760MHz for a speed grade 2 device.

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search