Build Your Own Domain-specific Solutions with RapidWright

Invited Tutorial

Chris Lavin and Alireza Kaviani
Xilinx Research Labs
San Jose, CA
chris.lavin@xilinx.com,alireza.kaviani@xilinx.com

ABSTRACT

As the complexity of programmable architectures increases with advances in silicon process technology, there is a growing need to extract greater productivity and performance from the tools. Due to their inherent reconfigurability, FPGAs are proving to be valuable targets for more efficient domain-specific architectures. However, FPGA implementation tools are designed for a broad set of applications.

In this paper we describe RapidWright, an open source framework that enables customized implementations for Xilinx FPGAs. RapidWright enables implementation tools that can take advantage of the great potential of domain-specific attributes—leading to greater productivity and performance. The focus of this paper is to provide an introductory reference of RapidWright and its use cases so that others may be empowered to adapt their implementations to their domain-specific applications.

CCS CONCEPTS

• **Hardware** → **Reconfigurable logic and FPGAs**; • **Computer systems organization** → *Reconfigurable computing*;

KEYWORDS

Domain-specific, Open Source, FPGA, Xilinx, Vivado

ACM Reference Format:
Chris Lavin and Alireza Kaviani. 2019. Build Your Own Domain-specific Solutions with RapidWright. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, Article 4, 9 pages. https://doi.org/10.1145/3289602.3293928

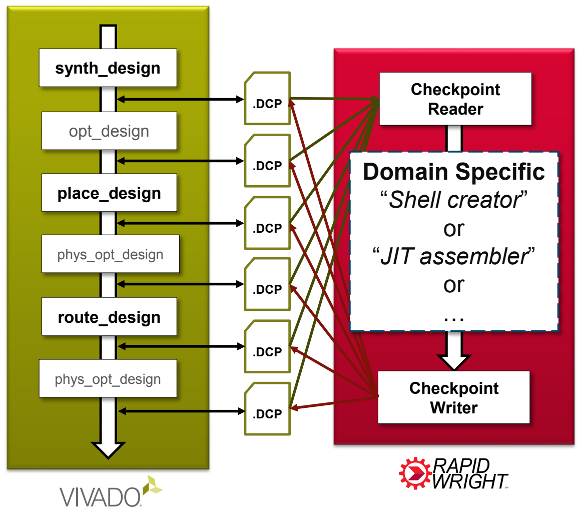


Figure 1: Vivado and RapidWright DCP Compatibility

1 INTRODUCTION

RapidWright [1] is an open source platform with a gateway to Xilinx's back-end implementation tools (Vivado) that raises the implementation abstraction while maintaining the full potential of advanced FPGA silicon. RapidWright works synergistically with Vivado through design checkpoints (DCPs, see Figure 1) to enable highly customizable implementations. Vivado can produce highly optimized implementations for key design modules to deliver the highest performance. RapidWright can then replicate, relocate and assemble these tuned modules to compose a complete application and preserve high performance.

RapidWright's native gateway to Vivado also sets the groundwork for an ecosystem aimed at further advancing FPGA tools. It empowers academic and industry researchers by combining the commercial credibility of FPGA tools with the agility of an open source framework, leading to innovative solutions that might not be feasible otherwise.

This paper serves as a supplemental reference to the RapidWright tutorial with an aim to provide some fundamentals about the framework and introductory use cases. In the remainder of this paper we describe RapidWright and its capabilities in Section 2, some example use cases in Section 3 and conclude in Section 4. Supplementary material on Xilinx architecture is included in Appendix A to help orient the reader regarding specific RapidWright constructs.

2 RAPIDWRIGHT STRUCTURE

RapidWright is implemented in Java and distributed with a foundational API library that provides access to design checkpoint (DCP) files and Vivado-compatible device models. A high-level diagram showing the organization of the project is shown in Figure 2. There are three core Java packages (groups of classes) within RapidWight: `device`, `edif` (logical netlist) and `design` (physical netlist) and this section describes the purpose and composition of each one.

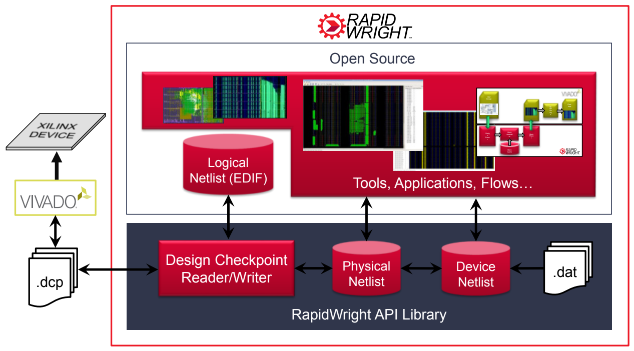


Figure 2: RapidWright Structure

2.1 Device Package

The device package contains classes and APIs that correspond to constructs in the silicon devices. The most prominent class in this package is the `Device` class, which makes available all of the architectural resources described in Appendix A. A device is supplemented by package, speed grade and temperature grade information through the use of a `Package` class. When a device is combined with its package and grade information, this uniquely identifies a Xilinx part, represented by the `Part` class.

The `Device` class is the top level object in RapidWright and has direct accessors to all other levels of hierarchy except for BELs as shown in the first row of Table 1. In contrast to the other two packages, the data provided in the `device` package is static. Most of the interaction between a user's design and the device occurs at the Tile, Site and BEL levels of hierarchy. The `BEL` class can be one of three kinds of non-routing objects in a `Site`: a Logic BEL, a Routing BEL and a Port (of the Site). This is designated by its class member enum of type `BELClass`. Most components within the device architecture are assigned an integer index. This helps to lower memory usage by eliminating the need to explicitly represent a component of the architecture with a dedicated object. It also helps by providing faster lookups. In some cases, such as `TileTypeEnum` and `SiteTypeEnum`, the index has been explicitly enumerated and an enum is used instead.

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

RapidWright provides the same inter-site routing resources as Vivado, namely `Wire`, `Node` and `PIP` objects (see second row of Table 1). These objects are generated on demand as there can be several millions of unique instances of each.

2.2 EDIF Package (Logical Netlist)

In Vivado, all designs post synthesis have a logical netlist that can be exported to the EDIF (Electronic Design Interchange Format) netlist format. Vivado also includes EDIF in the design checkpoint file format and 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 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.

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

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

Table 1: RapidWright and Vivado Device Object Model Reference

RapidWright Class	RapidWright Java API	Vivado Class Property	Vivado Tcl API
Device	`Device.getDevice(String partName)`	-	-
SLR	`Device.getSLR(int id)`	slr	`get_slrs -filter SLR_INDEX==$id`
ClockRegion	`Device.getClockRegion(String name)`	clock_region	`get_clock_regions $name`
Tile	`Device.getTile(String name)`	tile	`get_tiles $name`
Site	`Device.getSite(String name)`	site	`get_sites $name`
BEL	`Site.getBEL(String name)`	bel	`get_bels -of $site -filter NAME==$name`
PIP	`Device.getPIP(String name)`	pip	`get_pips $name`
Wire	`Device.getWire(String name)`	wire	`get_wires $name`
Node	`Device.getNode(String name)`	node	`get_nodes $name`
SitePIP	`Site.getSitePIP(BELPin input)`	site_pip	`get_site_pips $name`
SitePin	`Device.getSitePin(String name)`	site_pin	`get_site_pins $name`
int (SiteWire)	`Site.getSiteWireIndex(String name)`	site_wire	(Vivado GUI)
BELPin	`Site.getBELPin(String name)`	bel_pin	`get_bel_pins $name`

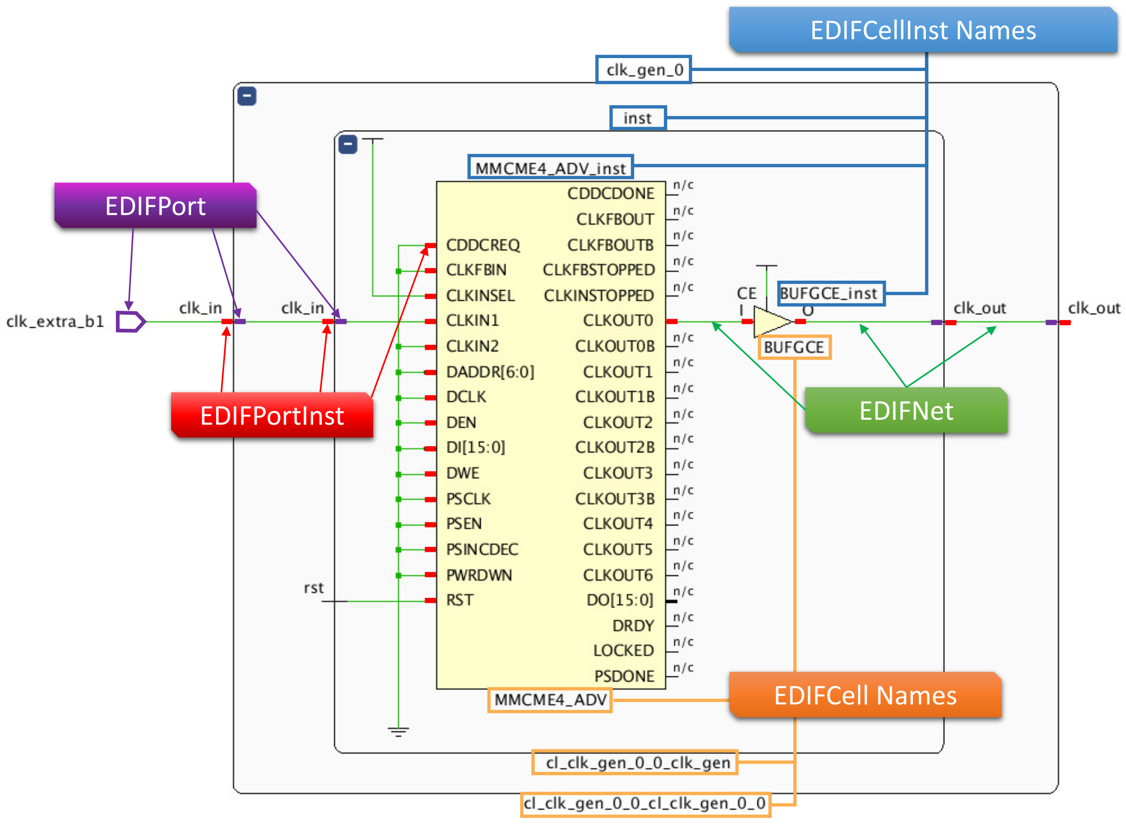


Figure 3: EDIF Data Structure Reference to Vivado Netlist View

references entering/leaving the cell). Figure 3 illustrates how RapidWright EDIF-based objects map to a Vivado netlist schematic view.

2.3 Design Package (Physical Netlist)

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

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

2.4 Cell (A BEL Instance)

At the lowest level, a RapidWright `Cell` maps a logical leaf cell from the EDIF netlist (`EDIFCellInst`) to a `BEL` as shown in Figure 4. The cell name is typically the full hierarchical logical name of the leaf cell to which it maps. A cell also maintains the logical cell pin mappings to the physical cell pin mappings (`BELPins`).

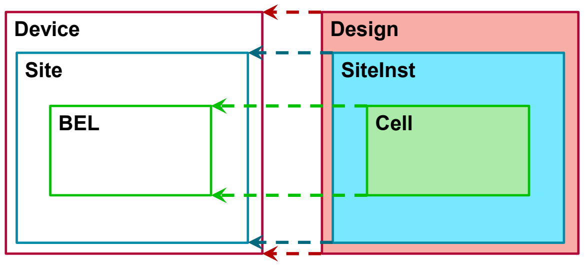


Figure 4: Shows mapping between BEL/Cell, Site/SiteInst and Device/Design.

2.5 SiteInst

Design representation and implementation in Vivado is BEL-centric (BELs and cells). The `SiteInst` keeps track of three major mappings/attributes:

(1) Map of all cells to BELs (placements in site)
(2) Activated Site PIPs (intra-site routing)
(3) Nets to Site Wires (intra-site routing)

Each `SiteInst` maps to a single, compatible site within a device. The `SiteInst` is configured to a type using a `SiteTypeEnum` that is either the primary type or an alternate site type of the host site. RapidWright also preserves the same Vivado “fixed” flag which is

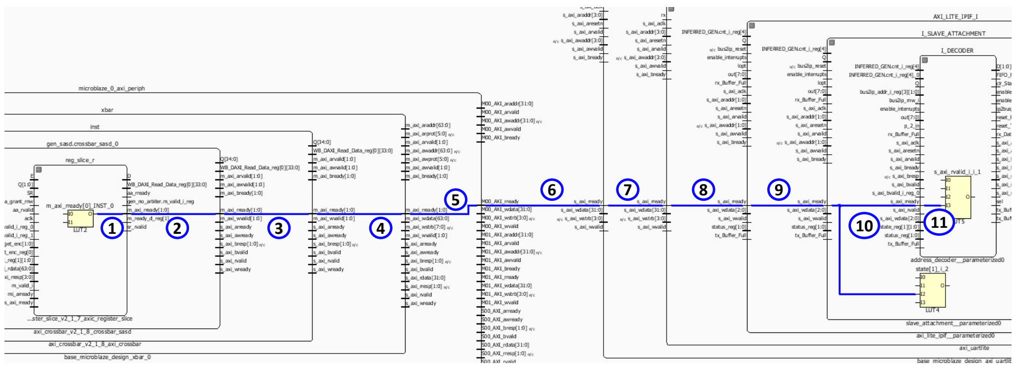


Figure 5: Logical netlist view of a particular physical net

used in certain situations to prevent components inside the site from being moved.

Routing nets inside of a site (intra-site) is different from routing outside of sites (inter-site) and the SiteInst maintains all relevant information concerning intra-site routing. Routing inside of a site must account for placed cells, their type and context. In general, when constructing placed and routed logic, it can be beneficial to compare SiteInst content from Vivado-generated implementations to ensure correctness. This can be done by loading placed and routed DCPs from Vivado into RapidWright and querying the respective SiteInst objects to establish patterns for site wire and site PIP usage.

Routing is accomplished inside a site through SitePIPs, which establish a connection through routing BELs and some logic BELs (such as LUTs). The SiteInst object in RapidWright maintains site PIP usage. By default, all site PIPs are turned off. If a SitePIP is added to the SiteInst then it is marked as being turned on or used.

2.6 Net

A Net in RapidWright contains the routing information to physically connect placed cells using device interconnect or PIPs. Many logical nets map to the same physical net, for example, consider the net depicted in Figure 5. This figure shows the logical netlist connection of three cells over one physical net. However, there are 11 separate logical nets (represented in RapidWright by a EDIFNet) in the logical netlist that must be traversed in order to make the connection. In contrast, Figure 6 shows one physical net for all logical nets.

The implementation of a physical net is stored as a collection of PIPs. PIPs connect nodes together and specify a path from a site pin source to one or more site pin sinks. These instances of site pins are represented by SitePinInst objects (instances of SiteInst objects). The rest of the physical net implementation (intra-site routing) is stored in a SiteInst where a path from site pins to BEL pins is described using annotated site wires and SitePIPs.

2.7 Module

A Module in RapidWright is a physical netlist container, which is a collection of SiteInst and Net objects that describe an abstract

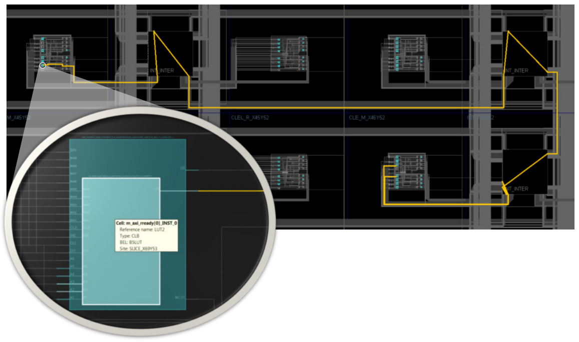


Figure 6: Physical netlist view of a particular physical net

definition of an implementation. This object is unique to RapidWright and is one of its enabling constructs that allows placed and routed information to be preserved, relocated and replicated. A module contains both the logical and physical netlist elements and corresponds to a hierarchical cell within a netlist. It is 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.

A RapidWright module is represented by the Module class in the design package. A module is a definition object whose SiteInst and Net objects specify a blueprint for a pre-implemented block that can potentially be ‘stamped’ out and relocated in valid locations around a device. The ModuleInst represents the instance object of a Module and is part of the implemented portion of a physical netlist.

2.8 Module Instance

A ModuleInst is an instance of a Module. Typically, definitions of a hierarchical cell are captured in a Module and then ‘stamped out’ using the module instance construct in a design. The placed and routed locations of the SiteInst and PIPs found in the Nets are relatively relocated according to the desired offset during instantiation or re-location. Modules typically pre-calculate all valid placement locations ahead of time and are stored with the module to make instantiation and placement fast.

ModuleInsts, like Modules, are a collection of SiteInst and Net objects. Each of these object names are prefixed with the name of the ModuleInst, for example, if a module had a SiteInst named "SLICE_X2Y2" and a Net named data_ready, a newly created module instance named "fred" would have counterpart SiteInst and Net objects called "fred/SLICE_X2Y2" and "fred/data_ready."

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

3 RAPIDWRIGHT USE CASES

RapidWright provides a unique set of capabilities not readily available using Vivado alone. Some of these capabilities include direct creation of placed and routed circuits, parameterizable circuit generators and module reuse through pre-implemented modules. This section briefly introduces these concepts as a primer of RapidWright capabilities.

3.1 Direct Synthesis of Placed and Routed Circuits

RapidWright is designed with sufficient capabilities to produce completely valid placed and routed circuits from scratch. It is not recommended to pursue this approach for large or complex designs, but it can be extremely useful in situations where a well-defined implementation is desired.

The circuit in Figure 7 is created, placed and routed using RapidWright code in Listing 1. Although this "hello, world" example is simple, it provides a small glimpse of the possibilities RapidWright has to offer. This circuit is a two-input AND gate packed into a LUT targeting the Zynq device on a PYNQ-Z1 board. It connects two button inputs to an output LED and will only illuminate if both buttons are pressed.

At runtime, the code is able to load the device model for a Zynq 7020 part, create the netlist, place the cells, route their interconnections and write out a DCP file in less than two seconds. Note that RapidWright provides APIs that both create cells in the netlist and places them on the device. For each Cell created, an EDIFCellInst is created and instantiated in the EDIFNetlist of the Design. This example also shows that intra-site routing and inter-site routing are separate APIs to allow for greater flexibility in implementation.

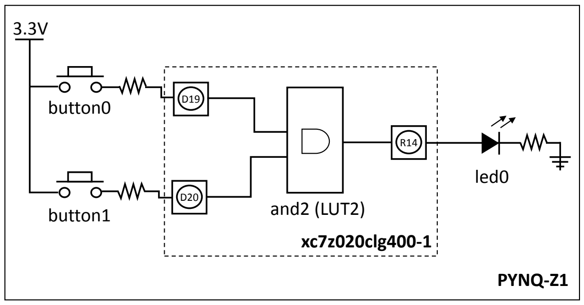


Figure 7: RapidWright "hello, world" Example

```
// Create a new empty design using the PYNQ-Z1 device part
Design d = new Design("HelloWorld",Device.PYNQ_Z1);

// Create and place all the design elements (LUT2, and 3 IOs)
String placementLoc = "SLICE_X100Y100/A6LUT";
String v = "LVCMOS33";
Cell and2 = d.createAndPlaceCell("and2", Unisim.AND2, placementLoc);
Cell button0 = d.createAndPlaceIOB("button0", PinType.IN , "D19", v);
Cell button1 = d.createAndPlaceIOB("button1", PinType.IN , "D20", v);
Cell led0    = d.createAndPlaceIOB("led0"   , PinType.OUT, "R14", v);

// Connect Button 0 to the LUT2 input I0
Net net0 = d.createNet("button0_IBUF");
net0.connect(button0, "O");
net0.connect(and2, "I0");

// Connect Button 1 to the LUT2 input I1
Net net1 = d.createNet("button1_IBUF");
net1.connect(button1, "O");
net1.connect(and2, "I1");

// Connect the LUT2 (AND2) to the LED IO
Net net2 = d.createNet("and2");
net2.connect(and2, "O");
net2.connect(led0, "I");

// Route intra-site nets (portions of net in a site)
d.routeSites();

// Route inter-site nets (between sites)
new Router(d).routeDesign();

// Save our work in a Design Checkpoint file
d.writeCheckpoint("HelloWorld.dcp");
```

Listing 1: RapidWright "hello, world" Code Example

3.2 Parameterizable Circuit Generators

RapidWright's ability to create fully placed and routed circuits from scratch enables a new class of design we call generators. Several parameterizable circuit generators are included with the RapidWright distribution. One significant example is the parameterizable SLR crossing generator which can produce a DCP solution within a few seconds. This SLR crossing generator targets UltraScale+ devices as they have the architectural capabilities that enable clocking techniques that achieve near-spec (>700MHz) performance (UltraScale and Series 7 devices, do not possess these capabilities).

The generator will create pairs of flip flops in a netlist for each crossing signal and will place them at the appropriate Laguna sites to leverage the dedicated super long line (SLL) interconnect paths. As mentioned in [1], using both dedicated RX and TX Laguna site flops will often produce hold time violations. RapidWright is able to circumvent this issue by routing the clock in such a way that all TX and RX flops are connected exclusively to the same clock arm. This enables a tuning of the clock delay at the common leaf clock buffer for each group of crossing signals in each direction respectively.

Additionally, the SLR crossing generator can potentially create a custom clock root for each SLR crossing group (crossings in the same clock region) to minimize the inter-SLR compensation timing penalty. By fabricating the netlist, placing the flops onto the dedicated RX and TX Laguna sites and custom routing the clock to tune leaf clock buffers and create clock roots, the generator is able to create a placed and routed DCP of an SLR bridge in a few seconds.

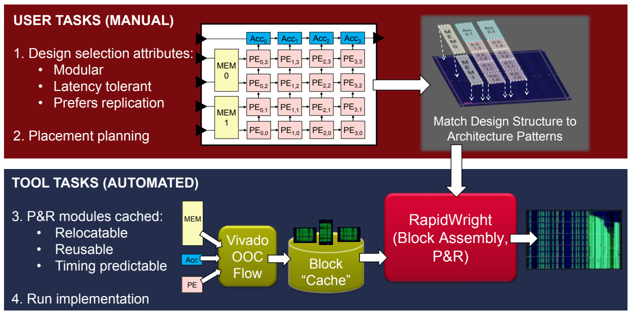


Figure 8: Pre-implemented Methodology

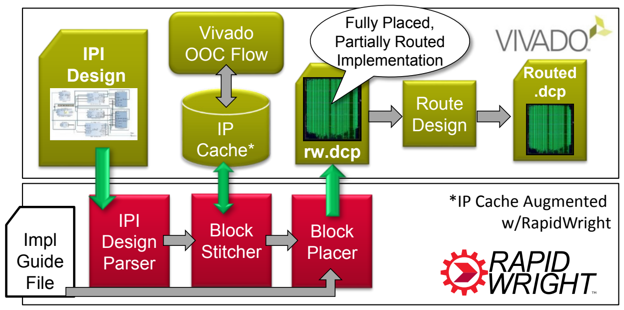


Figure 9: Pre-implemented Block Compilation Flow

3.3 A Modular Pre-implemented Methodology

One of the key attributes of RapidWright is the ability to capture optimized placement and routing solutions in 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, as design size grows, it is no longer practical to find near-optimal solutions within a short compile time. We show how to preserve and reuse high quality solutions in RapidWright with pre-implemented modules, and propose a methodology of how they can improve the overall system performance in a large design.

3.3.1 Pre-implemented Modules. Pre-implemented modules are self-contained netlist cells that contain relative placement and routing information (generally with a rectangular footprint) targeting a specific FPGA device. RapidWright generates pre-implemented modules by invoking Vivado to synthesize, place and route them out-of-context (OOC) of the original design. RapidWright then preserves and packages the placement and routing information from the OOC DCP as a RapidWright `Module` (see Section 2.7).

For a pre-implemented module to be reusable, it often needs to be area constrained with a pblock with the additional property `CONTAIN_ROUTING=1`. This ensures that placement and routing of the module is restricted to the respective rectangle, reducing its footprint such that it has a higher number of compatible placement locations across the device.

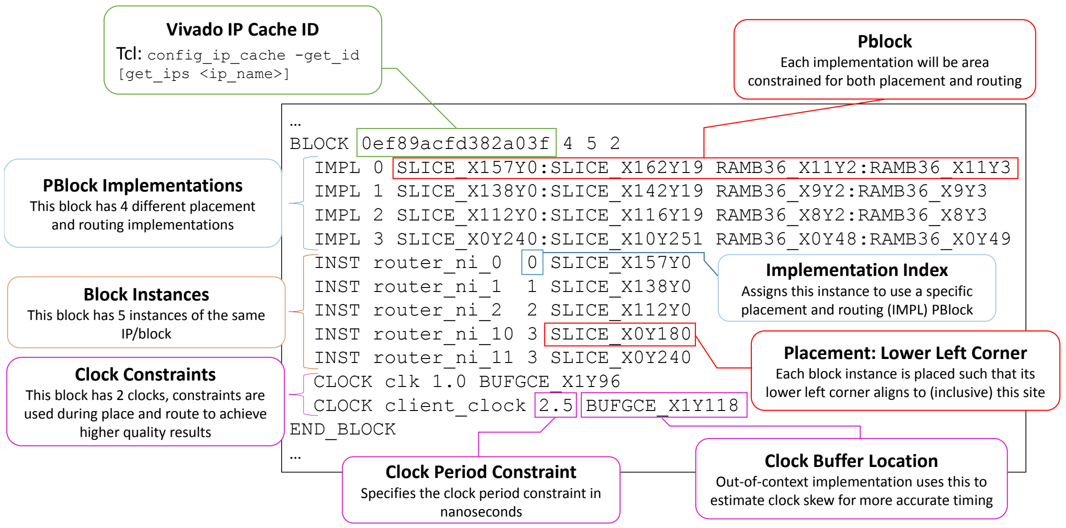
3.3.2 Design Strategy and Flow. RapidWright endows users with a new design vocabulary by caching, reusing and relocating pre-implemented blocks. We believe this to be an enabling concept and offer a high performance design strategy as depicted in Figure 8.

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

After the design architect has successfully restructured and modularized a design, step two of Figure 8 is followed. Here, the design architect creates an implementation guide file (see Section 3.3.3) that captures how best to map the modules of a design to the architecture of the target device. Specifically, pblocks (area constraints) 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 the 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, whose details are denoted in Figure 9. We leverage Vivado IP Integrator (IPI)[7] for design input. 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. The block stitcher reads the IP cache and populates the IPI design netlist. After stitching, the blocks are placed according to the implementation guide file from the design architect. Once all the blocks are placed, RapidWright creates a DCP file that is read into Vivado that completes the final routes.

3.3.3 Implementation Guide File. An implementation guide file (extension *.igf) allows the application architect to communicate all of the specific implementation customization aspects of the packing and placement phase. An example snippet of an implementation guide file can be seen in Figure 10.


Vivado IP Cache ID
Tcl: config_ip_cache -get_id [get_ips <ip_name>]
Pblock
Each implementation will be area constrained for both placement and routing
PBlock Implementations
This block has 4 different placement and routing implementations
Block Instances
This block has 5 instances of the same IP/block
Implementation Index
Assigns this instance to use a specific placement and routing (IMPL) PBlock
Placement: Lower Left Corner
Each block instance is placed such that its lower left corner aligns to (inclusive) this site
Clock Constraints
This block has 2 clocks, constraints are used during place and route to achieve higher quality results
Clock Period Constraint
Specifies the clock period constraint in nanoseconds
Clock Buffer Location
Out-of-context implementation uses this to estimate clock skew for more accurate timing

```
...
BLOCK 0ef89acfd382a03f 4 5 2
  IMPL 0 SLICE_X157Y0:SLICE_X162Y19 RAMB36_X11Y2:RAMB36_X11Y3
  IMPL 1 SLICE_X138Y0:SLICE_X142Y19 RAMB36_X9Y2:RAMB36_X9Y3
  IMPL 2 SLICE_X112Y0:SLICE_X116Y19 RAMB36_X8Y2:RAMB36_X8Y3
  IMPL 3 SLICE_X0Y240:SLICE_X10Y251 RAMB36_X0Y48:RAMB36_X0Y49
  INST router_ni_0  0 SLICE_X157Y0
  INST router_ni_1  1 SLICE_X138Y0
  INST router_ni_2  2 SLICE_X112Y0
  INST router_ni_10 3 SLICE_X0Y180
  INST router_ni_11 3 SLICE_X0Y240
  CLOCK clk 1.0 BUFGCE_X1Y96
  CLOCK client_clock 2.5 BUFGCE_X1Y118
END_BLOCK
...
```

Figure 10: Implementation Guide File (*.igf) Example

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.

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.

The SUB_IMPL is an optional construct that allows finer-grained pblocks to be 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. For example, if a particular IP is tall and narrow and there are specific cells that need to be placed at the top and bottom, the SUB_IMPL construct can be used to pblock the top and bottom specific cells in sub pblock of the overall implementation.

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.

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.

4 CONCLUDING REMARKS

We have provided an introductory overview and use cases for RapidWright. RapidWright enables an implementation vocabulary that lays the ground work for next generation domain-specific tools targeting FPGAs. As FPGAs present a valuable platform for domain-specific architectures, the tools' productivity and performance will become even more critical to the success of a project. We invite industry and academic researchers to help us build a new generation of domain-specific tools that will further capitalize on the potential of FPGAs.

For more examples, documentation and tutorials on RapidWright, please visit www.rapidwright.io.

A APPENDIX: XILINX ARCHITECTURE

RapidWright is an implementation-centric framework targeting Xilinx FPGAs and to use it effectively, an understanding of Xilinx FPGA architecture will be needed. RapidWright presents the same constructs and device representations found in Vivado's device model. A cross-reference between RapidWright and Vivado objects/APIs is shown in Table 1. There are there are six major levels of hierarchy used to implement logic as shown in the first row of Table 1: Device, SLR, ClockRegion, Tile, Site and BEL. These six logic hierarchy levels are also illustrated in Figure 11. There are also several wiring and interconnect constructs related to routing listed in row 2 of Table 1: PIP, Wire, Node, SitePIP, SitePin, SiteWire and BELPin. With the exception of SitePIP, these are illustrated in Figure 12. The remainder of this section will briefly describe each of the six logical hierarchy objects with routing objects described in their context.

A.1 BEL (Basic Element of Logic)

The atomic unit of Xilinx FPGAs is a basic element of logic (BEL). There are two kinds of BELs, Logic BELs and Routing BELs. A Logic BEL is a configurable logic-based site that can support the implementation of a design cell (such as a LUT or flip-flop). Each BEL can support one or more types of UNISIM cells (UNISIM cells are described in [4] for Series 7 devices and [6] for UltraScale devices). The mapping between a leaf cell in the netlist and a BEL site is

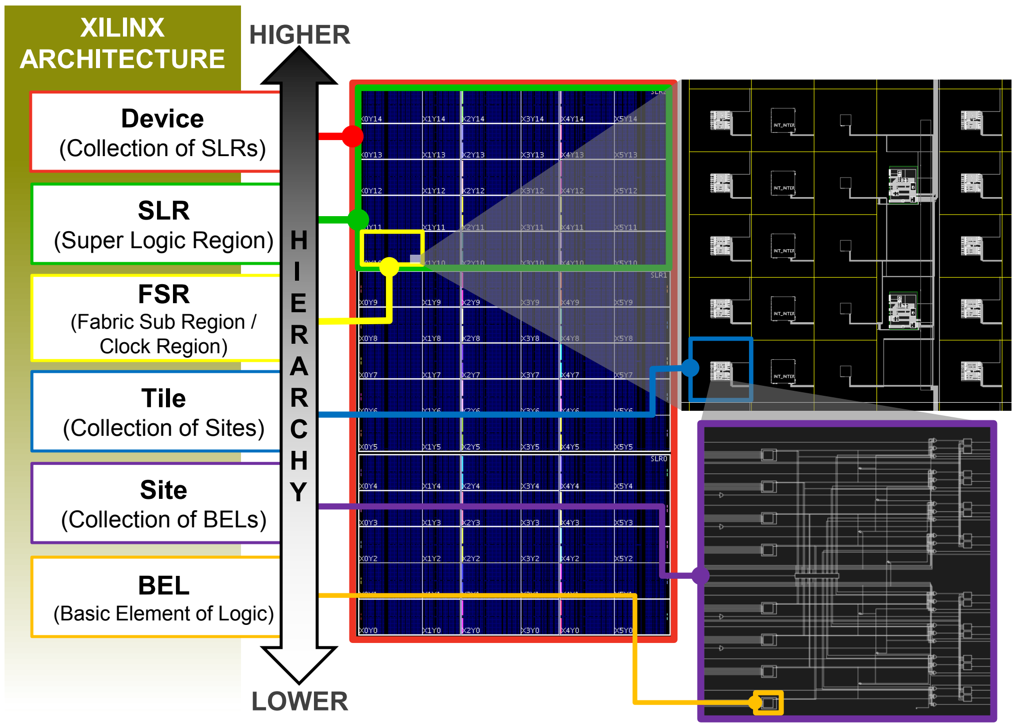


Figure 11: Xilinx FPGA Architecture Hierarchy

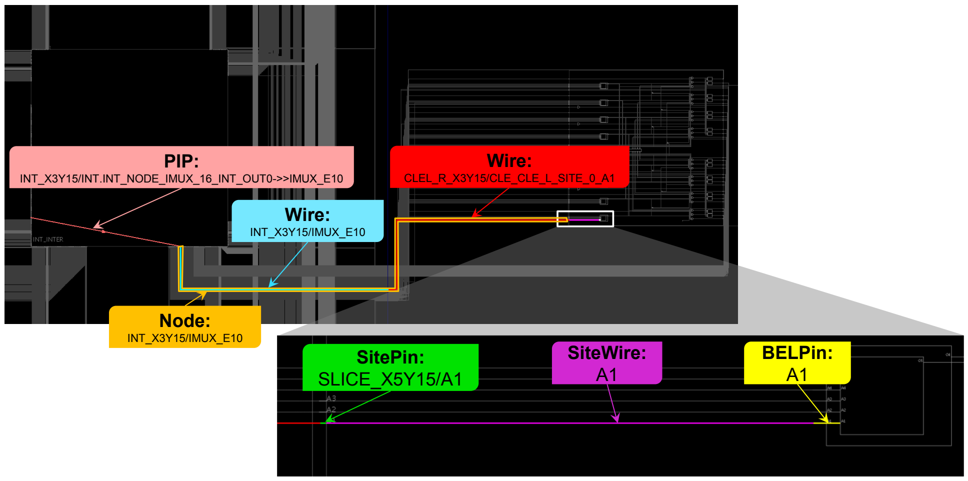


Figure 12: Intersite and Intrasite Routing Resources

referred to as the "placement" of the cell. Non-leaf cells represent hierarchy of the netlist and do not require placement. 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 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). However, some routing BELs do have optional inversions.

BELs have input and output pins and configurable connections that connect an input pin to an output pin. These BEL-based configurable connections are called site PIPs (Programmable Interconnect Points). 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 for the site PIP to be usable. These site PIPs, when implemented in logic BELs (such as a LUT), are called "route-thrus." When routing a design, it is sometimes necessary to route through unused LUTs (or other BELs) using site PIPs to complete a route.

A.2 Site

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

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

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

A.3 Tile

A collection of sites is packaged into a tile, although several tiles do not have sites. 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 designed to abut one another when laid down to construct an FPGA device.

Unlike sites and BELs, tiles do not have user visible pins. Instead, tiles contain uniquely-named wires that can connect to site pins or PIPs. In the context of a tile, PIPs connect two tile wires together. 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 Figure 13.

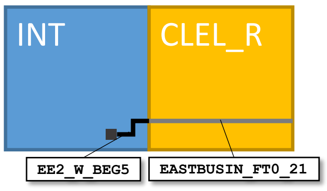


Figure 13: Two wires in abutting tiles

As there are no pins on tiles, instead cross-tile connectivity is represented by a node. A node is a collection of electrically connected wires that spans one or more tiles. Figure 14 shows how four wires in four tiles abut to form a node.

A.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. Xilinx uses a column-based architecture where each column of tiles are generally of the same type. In the UltraScale architecture, all FSRs are 60 CLBs (common logic block tiles) tall, but their width will vary depending on the mix of tile types used in its construction.

Clock routing and distribution lines are represented at the same granularity as clock regions. In UltraScale architectures, there are

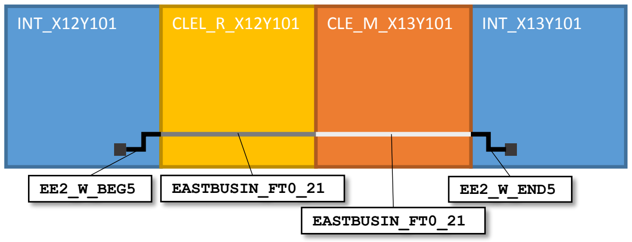


Figure 14: A node composed of four wires in tile context

24 horizontal routing tracks, 24 vertical routing tracks, 24 horizontal distribution tracks and 24 vertical distribution tracks per clock region. These routing and distribution tracks abut to tracks in neighboring clock regions to form the device clock network resource set. Additional information specific to clocking resources can be found in [2] for Series 7 devices and [3] for UltraScale devices.

A.5 SLR (Super Logic Region)

A super logic region (SLR) is a 2D grid of FSRs. This level of hierarchy is only relevant on devices that use stacked silicon interconnect technology (SSIT – also known as 2.5D), essentially a packaging of multiple die together with a silicon interposer. Each die in a multi-die device is an SLR.

In order for logic to communicate between SLRs, the UltraScale architecture employs special "Laguna" tiles in the FSRs neighboring the abutment of two SLRs. Laguna tiles have dedicated flip flop sites to aid in crossing the SLR divide.

A.6 Device

At the highest level of Xilinx architecture is the device. This encapsulates any and all SLRs present. The device object in Vivado is implicit (not directly referenced) but only one device can be loaded at a time. The core object in RapidWright is the `Device` class for any Xilinx device as described in Section 2.

REFERENCES

[1] C. Lavin and A. Kaviani. 2018. RapidWright: Enabling Custom Crafted Implementations for FPGAs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, https://doi.org/10.1109/FCCM.2018.00030, 133–140.
[2] Xilinx, Inc. 2018. *UG472 (v1.14): 7 Series FPGAs Clocking Resources User Guide.* Xilinx, Inc. https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf.
[3] Xilinx, Inc. 2018. *UG572 (v1.7): UltraScale Architecture Clocking Resources User Guide.* Xilinx, Inc. https://www.xilinx.com/support/documentation/user_guides/ug572-ultrascale-clocking.pdf.
[4] Xilinx, Inc. 2018. *UG953: Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide.* Xilinx, Inc. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug953-vivado-7series-libraries.pdf.
[5] Xilinx, Inc. 2018. *UG973 (v2018.1): Vivado Design Suite User Guide Release Notes, Installation and Licensing.* Xilinx, Inc. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug973-vivado-release-notes-install-license.pdf.
[6] Xilinx, Inc. 2018. *UG974: UltraScale Architecture Libraries Guide.* Xilinx, Inc. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug974-vivado-ultrascale-libraries.pdf.
[7] Xilinx, Inc. 2018. *UG994 (v2018.2): Vivado Design Suite User Guide Designing IP Subsystems Using IP Integrator.* Xilinx, Inc. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug994-vivado-ip-subsystems.pdf.