Vivado Design Interface: An Export/Import Capability for Vivado FPGA Designs

Thomas Townsend and Brent Nelson
NSF Center for High-Performance Reconfigurable Computing (CHREC)
Dept. of Electrical and Computer Engineering
Brigham Young University
Provo, UT, 84602, USA
Email: {thomastownsend523, brent_nelson}@byu.edu

***Abstract*—Research tools targeting commercial FPGAs have most commonly been based on the Xilinx Design Language (XDL). Vivado, however, does not support XDL, preventing similar tools from being created for next-generation devices. Instead, Vivado includes a Tcl interface that exposes Xilinx's internal design and device data structures. Considerable challenges still remain to users attempting to leverage this Tcl interface to develop external CAD tools. This paper presents the Vivado Design Interface (VDI), a set of file formats and Tcl functions that address the challenges of exporting and importing designs to and from Vivado. To demonstrate its use, VDI has been integrated with RapidSmith2, an external FPGA CAD framework. To our knowledge this work is the first successful attempt to provide an open-source tool-flow that can export designs from Vivado, manipulate them with external CAD tools, and re-import an equivalent representation back into Vivado.**

I. Introduction

Over the past few decades, a significant amount of CAD tool research for FPGA-based systems has been pursued outside the confines of vendor software. The most common approach to creating external tools has been to use the extremely successful VPR/VTR CAD suite [1] [2]. VTR provides methods for mapping designs to theoretical FPGA architectures described in architectural definition files. Using these architecture definitions, researchers have been able to explore both new FPGA architectures as well as CAD algorithms for packing, placement, and routing on those architectures.

Targeting commercial FPGA devices has traditionally been more difficult. In recent years, the Xilinx Design Language (XDL) has been a popular solution for creating external CAD tools that target Xilinx parts supported in ISE. Tools that target other vendor parts have been created [3], but are less common. With the release of Vivado, however, Xilinx no longer supports XDL, making external tools and frameworks that rely on the interface incompatible with next-generation Xilinx devices (such as UltraScale, Ultrascale+, and beyond).

Vivado provides access to its design and device data structures through a Tcl API exposed to the user. The same information that was contained within XDL can now be extracted through Tcl API calls, but there are many challenges associated with using Tcl for this purpose. This paper presents the Vivado Design Interface (VDI), a set of file formats and Tcl functions that address the challenges of exporting and importing designs to and from Vivado. VDI is an XDL-like interface that can be used with external CAD tools targeting Vivado devices. To demonstrate this, VDI has been integrated with RapidSmith2 [4], a Xilinx FPGA CAD tool framework whose data structures closely match those of Vivado. This paper is intended to give a high-level overview of VDI. A more complete and detailed description is given in [5].

The remainder of the paper is organized as follows. Section II provides background on XDL and its many applications. Section III describes a subset of the challenges encountered with Vivado's Tcl interface to export and import design information. Section IV describes VDI, the main contribution of this work. Section V discusses how to integrate VDI with external tools, using RapidSmith2 as an example. Section VI gives the results of several experiments run using VDI. Section VII concludes the paper.

II. Background

The Xilinx Design Language is a command line interface into Xilinx's ISE tool suite. Using a single command, `xdl`, both device and design information can be extracted from ISE for external use. Device specific information is exported via XDLRC files, which contain a detailed listing of all the physical components inside of a Xilinx FPGA. This includes the tiles, sites, BELs, and routing resources (without timing information) available in the specified part. Design data is exported via XDL files. XDL files contain both the logical portion of a design, as well as placement and routing information. The combination of XDLRC and XDL files make it possible to perform design manipulations outside the confines of vendor tools. `Xdl` also allows a modified XDL netlist to be re-imported into ISE's internal netlist structure.

Most research experiments and open-source CAD tools targeting Xilinx devices have been built upon XDL. Frameworks such as RapidSmith [6] and Torc [7] provide easy-to-use APIs to modify XDL netlists in a variety of useful ways. The contributions of projects leveraging XDL have been many. [8] and [9] create partial reconfiguration frameworks, capable of

This work was supported by the I/UCRC Program of the National Science Foundation under Grant No. 1265957.

swapping logic segments at runtime. [10] and [11] look at ways to decrease the implementation times of FPGA designs and increase designer productivity. [12] and [13] look to create reliable, fault-tolerant FPGA systems. And [14] tries to bridge the gap between the VPR CAD flow with real Xilinx devices. Many significant contributions have also been made in other areas of FPGA research such as security and debugging.

Clearly, analyzing and modifying FPGA designs outside of vendor tools has proven useful. As previously stated, however, Vivado does not support XDL. This leaves two possibilities for CAD tools targeting Vivado devices: (1) create a framework to facilitate design manipulations written directly in Vivado's Tcl interface, or (2) create an XDL-alternative for Vivado. In previous work, the authors of `Tincr` [15] examined both options and concluded that option (2) is preferable because of Tcl limitations. They also demonstrated that it is possible to extract design and device information out of Vivado with Tcl commands, but this extraction was far from complete. For devices, the internal routing structure of sites was not included. For designs, the extraction was simply a proof-of-concept since several important implementation details of a design were omitted from the exported format. VDI builds upon the initial work done with `Tincr` to create a complete CAD flow.

III. Vivado's Tcl Interface

Vivado's Tcl interface gives users the ability to script design flows, set constraints, and perform low-level design modifications. It is a powerful addition to the Xilinx tool suite, but is not well suited for the creation of custom CAD tools. This is for three reasons in particular:

- Tcl, an interpreted language, is slow. Compiled or managed runtime systems are better options for performance.
- Vivado's Tcl interface does not manage memory well. A simple Tcl script can cause the memory usage of Vivado to grow quite large.
- Tcl does not natively support higher-level programming constructs, making it more difficult to implement complex algorithms (such as PathFinder).

It is clear that the Tcl API is not intended to be used for large-scale design manipulations. This motivates our approach: use the Tcl interface to extract device and design information from Vivado, and format the information for external CAD tools.

Importing and exporting information through the Tcl interface results in several unique challenges. The remainder of this section gives an overview for a subset of the challenges that need to be addressed by a Vivado interface tool.

A. Cell Placement Order

When a cell is placed in Vivado using Tcl API calls, Vivado automatically configures the routing resources inside of the corresponding site based on the cell's connections. Because of this behavior it is possible to have a valid cell placement in Series7 architectures, but to place the cells in an order that causes an illegal routing configuration (i.e. a routing mux is optionally being used by a net, but is required by a different net of a cell that has just been placed). This primarily affects sites of type `SLICEL` and `SLICEM`, due to their more complex internal routing. Experimentation has shown that the proper placement order for groups of cells mapped to `SLICE` sites is as shown in Figure 1. If the required placement order is not followed when recreating a design, internal routing conflicts will occur that Vivado will be unable to resolve.

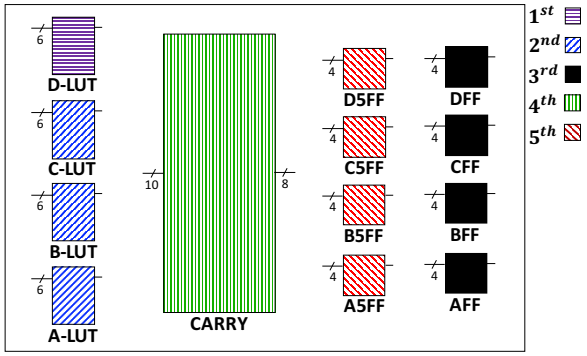

Fig. 1: Cell placement order for `SLICEL` and `SLICEM` sites

B. Macro Primitives

Xilinx-specific library cells fall into two categories: *leaf* primitives and *macro* primitives. Leaf primitives are the basic building blocks of a Xilinx netlist, such as LUT and flip flop cells. Macro primitives are hierarchical, and are made up of one or more leaf primitives wired together to perform a specific function. For example, the `RAM128X1D` macro is a 128-deep by 1-bit wide LUT RAM that consists of four `RAMD64E` and two `MUXF7` leaf primitives. A flattened Vivado netlist will contain a mixture of leaf and macro primitives.

Vivado uses EDIF to export design netlists. Regardless of whether the design is flattened during synthesis, macro primitives will remain in the generated EDIF file. However, the internal structure of each macro is *not* included. To fully reconstruct designs in external tools, the internal contents of macro primitives must be supplied in addition to the EDIF.

C. Directed Routing Strings

The structure of a physically routed net in Vivado is represented by a *directed routing string*. This string represents the tree structure of a physical route by using nested brackets ("{") as branches. It is stored in the `ROUTE` property associated with the corresponding logical net. By default, `ROUTE` strings contain only partial wire names. A wire in Vivado is uniquely identified by a tile name combined with a wire name. `ROUTE` strings only contain the wire name portion, making them ambiguous in some situations. That is, a source wire can connect to two sink wires with the same name, but in different tiles. This makes it impossible to reliably reconstruct routes in external tools using `ROUTE` strings.

D. Configurable Pin Mappings

When a cell is placed onto a physical BEL, the pins of the cell are mapped onto the pins of the BEL. However, this mapping changes based on *how the cell is configured*. For example, the pin mappings for a BRAM cell with a data width of 72 bits differ from the pin mappings of a BRAM cell with a data width of 9 bits, even if they are placed at the same physical location. When exporting placement information from Vivado, it is not enough to only give the cell-to-BEL mappings. Cell-pin mappings are also required.

E. Incomplete Tcl Representation

Some aspects of Vivado devices and designs are inaccessible through the Tcl interface. One example is that the internal routing structure of a site cannot be programmatically determined through Tcl API calls. Most objects have an associated *get_* method (such as *[get_tiles]*), but there is no such method for site wires. This indicates that the routing structure of sites must be created a different way.

Another example involves alternate site pins for Series7 architectures. In Xilinx FPGAs all physical sites have a default type, but some can be configured to be one of many types. For example, a `BUFGCTRL` site can also be configured to be of type `BUFG`. When the type of a site is changed in Vivado, the site pins attached to the site may get renamed to match the correct type. However, the Tcl function *[get_site_pins -of $site]* will always return the default site pin names, even if the type of the site has changed.

F. Logical and Physical Representation Mismatches

Besides their use in implementing logic equations, LUT BELs can also act as signal routethroughs and static signal sources. Figure 2 shows an example of each of these alternate uses. A routethrough LUT passes the signal on an input pin directly to the output pin. A static source LUT outputs either 0 or 1. In both cases, the LUT is simply a physical implementation detail of the design, and *is not represented in any way by the logical netlist*.

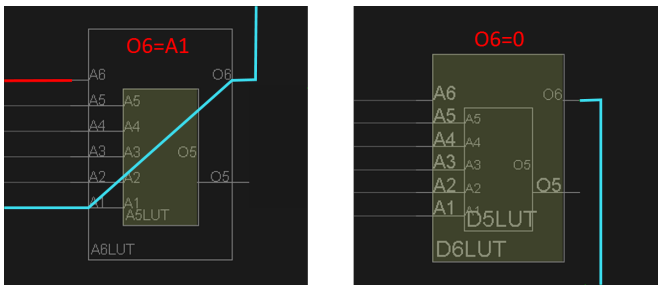

Fig. 2: Routethrough (left) and static source (right) LUT

A second mismatch involves power (VCC) and ground (GND) nets. In general, a design can have more than one VCC net in the logical netlist. Each of these nets should have their own unique `ROUTE` string once the design is routed (based on which wires they use), but this is not the case with Vivado. Instead, they all have the *same* `ROUTE` string which corresponds to a combination of *all* VCC nets. This also applies to GND nets. There are two possible solutions to resolve this discrepancy. The first is to merge all logical VCC (or GND) nets in the netlist into one net, matching the merged physical routing. The second is to partition the physical information so that each logical net has a unique `ROUTE` string.

In summary, there are a number of challenges when using Vivado's Tcl interface to export and import device and design information. Only a representative subset of challenges have been presented in this section. A complete export/import solution should abstract the low-level details of these issues away from the end user to facilitate the creation of custom CAD tools for Vivado.

IV. Vivado Design Interface

This section introduces the Vivado Design Interface (VDI), a Vivado export/import capability that provides solutions to handle the many challenges discussed in section III. As Figure 3 shows, VDI defines a set of file formats that can represent Vivado designs and devices for external use. Xilinx devices are represented with a XDLRC file and a series of XML files that add additional useful information about a device. Xilinx designs are represented with RSCP and TCP files.

VDI has been added to `Tincr` [15], and is available at https://github.com/byuccl/tincr. For each intermediate file shown in Figure 3, a `Tincr` command has been created to parse or generate the file. VDI users can utilize these commands to generate device and design files that can be loaded into the data structures of external tools. After an algorithm is run in an external tool, a modified design can be formatted into a TCP and imported into Vivado to complete the implementation flow. VDI currently supports fully-flattened designs from Vivado version 2016.2. The remainder of this section details each `Tincr` component of VDI and its purpose.

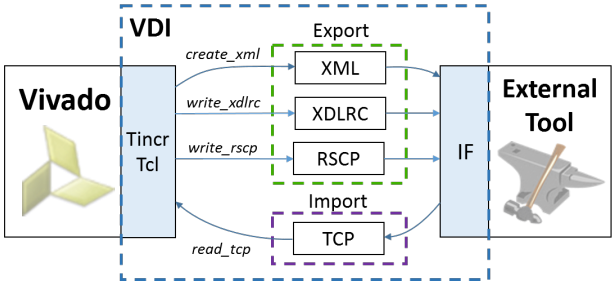

Fig. 3: The Vivado Design Interface

A. XDLRC Files

The original `Tincr` distribution included a command to generate XDLRC files for Vivado devices. However, these files are missing a key part: the primitive definition section. A primitive definition details all pins, BELs, and routing muxes *inside* of a site, and how they are connected. As described in section III, all of these physical components can be extracted

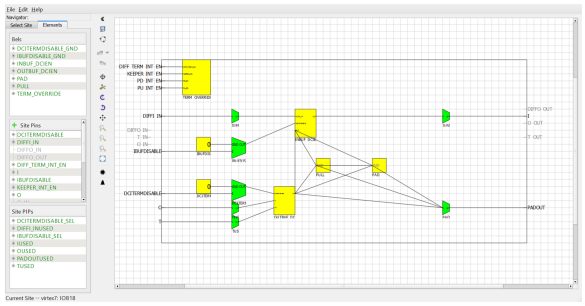

Fig. 4: VSRT GUI for creating primitive definition connections

through the Tcl interface *except* the site wires. To give a complete device description for external tools, the **intrasite** connections need to be generated another way.

Our solution is the Vivado Subsite Routing Tool (VSRT), shown in Figure 4. Using VSRT a user can bring up a primitive site in both Vivado's device browser and the VSRT GUI, and manually draw the site wires. Once the connections have been drawn, VSRT will generate the required primitive definition file automatically which can be integrated with the corresponding XDLRC. It is important to note that this is a time-consuming process, but only needs to happen once for each device family (usually once per series). Also, many of the sites can be done automatically without manual intervention.

B. Vivado Import Format

To import designs into Vivado, the Tincr Checkpoint (TCP) format introduced in the original `Tincr` distribution is used. TCPs use XDC constraint files which support a subset of Vivado Tcl commands. There are four primary parts of a TCP:

- netlist.edf - EDIF netlist
- contraints.xdc - User defined XDC constraints
- placement.xdc - Placement XDC constraints
- routing.xdc - Routing XDC constraints

Vivado has a dedicated Tcl command *[read_xdc]* that can parse and apply XDC constraint files to a design. This command is significantly faster than running each individual constraint which is why XDC files are chosen. XDC constraints are processed sequentially from the start of the file, meaning that the *placement.xdc* file of a TCP needs to be properly formatted according to the placement rules in subsection III-A.

There are some limitations to importing designs into Vivado using XDC. One example is that LUT BELs cannot be configured as routethroughs using Tcl commands. Another is that internal cells of a macro primitive cannot be explicitly placed within XDC. Because of these limitations it is the responsibility of external CAD tools to handle some tasks while generating TCPs. A more detailed description of these tasks, along with solutions for each, is discussed in section V.

C. Vivado Export Format

RapidSmith Checkpoints (RSCP) are used to export design information out of Vivado at any stage of implementation (post-synthesis, post-place, or post-route). RSCPs are created to be easily parse-able, and include *all* implementation details of a design. It is important to note that the name "RapidSmith Checkpoint" does not make this export format exclusive to RapidSmith. It was simply chosen to match the external framework that was used to test the interface. Any external tool can use RSCPs. A brief overview of the files included in a RSCP is given in this section.

1) netlist.edf: An EDIF netlist representing the logical portion of a design. It details all cells, nets, and ports within a design (with their corresponding properties), and is generated from Vivado using the Tcl command *[write_edif]*.

2) constraints.rsc: Contains all constraints on a design. This includes the clock frequency, top-level port to package pin mappings, and other physical implementation properties.

3) placement.rsc: Contains all placement information for a design. This includes which package pin every port is mapped to, which BEL each cell is placed on, and the logical-to-physical pin mappings for each placed cell-pin. If a design has not yet been placed, this file will be empty.

4) routing.rsc: The *routing.rsc* file is the most complex file in a RSCP, and stores all routing information for a Vivado design. This includes:

- The intrasite routing configuration for each site in the form of site pips (routing muxes).
- A list of LUT routethroughs. This includes the input pin, output pin, and BEL of each routethrough.
- A list of BEL LUTs that are VCC or GND sources
- A list of PIPs used in each routed net. To overcome the ROUTE property ambiguities described in subsection III-C, each PIP in the list includes the tile names and relative names for the wires connected together through the PIP. An algorithm to reconstruct a routing tree from this list is described in section V.
- The **merged** physical routing information for VCC and GND nets. This includes all VCC/GND drivers.

If a design has not yet been routed, only the intrasite routing information will be included in this file. If the design has not yet been placed, this file will be empty.

5) macros.xml: Contains template information about macro cells in a Vivado design that were not fully flattened, and do not exist in the default Vivado cell library (described next).

D. Cell Library XML

A Vivado netlist is composed of Xilinx primitives that have been instanced from template library cells. To do any netlist manipulation, such as adding cells to a design, detailed information about these library cells is required by external tools. This includes the name, pins, properties, and valid placement locations for each cell. VDI stores this information into a `Cell Library` XML. External CAD tools can parse the XML into their own data structures to be used for logic manipulation and design reconstruction. Information about the available macro primitives is also included.

There is, however, one problem when generating a `Cell Library`. The Tcl function *[get_lib_cells]* is used to get a list of all library cells that should be included, but this function

does not return all macro cells that could possibly appear in a flattened Vivado netlist. For example, the macro primitive `RAM128X1D` referenced in subsection III-B is not returned from *[get_lib_cells]*. A part of the RSCP generation process is to create the macro XML for these missing library cells, and store them into a *macros.xml* file. The `Cell Library` XML combined with a *macros.xml* file provides a full representation of all cells in a given design.

E. Family Info XML

XDLRC device files only contain the basic physical information of a device. They do not include other relevant information such as:

- Possible alternate types of a site
- Pin renamings for alternate site types
- Compatible sites (the same group of cells can be placed on multiple site types)
- The distinction between logic BELs and routing muxes
- BELs that can be used as routethroughs

VDI stores this information into a `Family Info` XML file. External CAD tools can parse this XML to create a more complete device representation

V. Integrating VDI with RapidSmith2

Using VDI, RapidSmith2 [4] now supports importing and exporting Vivado designs. RSCPs can be parsed and loaded into RapidSmith2 data structures, where CAD tools can operate on the design. Also, TCPs can be generated from RapidSmith2 and then imported back into Vivado. Figure 5 depicts all possible design flows using RapidSmith2 in conjunction with VDI. The RapidSmith2 project is available for checkout and collaboration at https://github.com/byuccl/RapidSmith2.

Integrating VDI with RapidSmith2 demonstrates that VDI provides all required information to recreate Vivado designs in external tools. However, as Figure 3 suggests, there are some tasks that external tools are responsible for to create a complete design flow. The remainder of this section enumerates these tasks, and how RapidSmith2 chooses to handle them.

A. Importing RSCPs

1) VCC and GND Nets: As described in subsection III-F, the logical representation of VCC nets in Vivado does not match the physical representation. For simplicity, RapidSmith2 chooses to collapse all logical VCC nets in the EDIF netlist into one global VCC net during design import. An identical process occurs for GND nets. This is the suggested approach for external tools that use VDI.

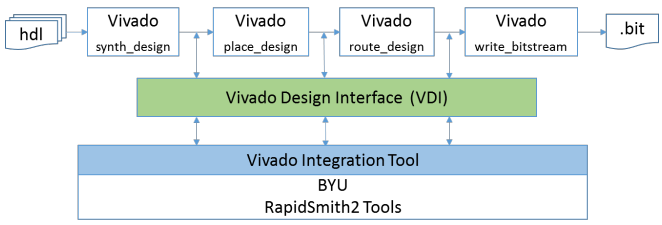

Fig. 5: RapidSmith2 Design Flows

2) Recreating Routing Trees: As described in subsection IV-C, a routed net is exported in a RSCP as a list of PIPs with no information about branching. Given a source wire, the routing tree of a net can be reconstructed by using these PIPs. A basic algorithm for this (shown in Listing 1) uses a simple breadth-first search through the wires of the device, connecting wires whose PIPs are enabled. Nets with multiple drivers (such as VCC) call this function for each driver.

Listing 1: Algorithm to Import Routing Tree

```
Set<PIP> usedPipSet = all PIPs in route (from RSCP)
Wire source = first wire in route;

Queue<Wire> queue = new Queue();
queue.enqueue(source);

while(!queue.isEmpty())
  Wire currentWire = queue.dequeue();
  for each pip sourced by currentWire
    if (usedPipSet.contains(pip))
      wire.makeConnection(pip.getSinkWire());
      queue.enqueue(pip.getSinkWire());

return source;
```

B. TCP Export

1) Routethrough LUTs: As described in subsection IV-B, LUT BELs in Vivado cannot be configured in Tcl. Before generating a TCP, external tools must configure routethroughs themselves. RapidSmith2 does this by first identifying all BELs that are being used as a routethrough. For each routethrough, a `LUT1` cell is created and inserted into the RapidSmith2 netlist and then placed onto the corresponding BEL. This forces Vivado to recognize the LUT BEL as a routethrough, but doesn't affect the final circuit behavior.

2) Macro Cell Placement: As discussed in subsection IV-B, internal cells to macro primitives cannot be placed in XDC files. There are two possibilities to alleviate this. The first is to fully flatten the design netlist when exporting a design (remove the macros completely). The second, and more complex option, is to support relatively placed macros (RPMs) in the external tool. Supporting RPMs involves understanding all possible placement configurations for a macro (and its internal cells), and how to place macro cells through XDC. RapidSmith2 implements the first option — macro cells are supported, but are ultimately flattened on design export.

3) Sorting Cells: As described in subsection III-A, cells mapped to `SLICEL` or `SLICEM` sites need to be placed in a very specific order to prevent routing conflicts. When generating the *placement.xdc* of a TCP, external tools need to ensure that placement constraints for these cells match the required order shown in Figure 1. RapidSmith2 uses a bin sorting algorithm to sort the cells in the required order before generating a TCP.

VI. Results

To show the validity of VDI, several different experiments and tests were run using RapidSmith2 for both Series7 and

UltraScale architectures. A set of 21 benchmarks which range in size from 59 cells and 80 nets, to 68,376 cells and 68,451 nets were used primarily for testing. 11 of the benchmarks targeted the Artix7 part *xc7a100-tcsg324*, and 10 of the benchmarks targeted the Kintex UltraScale part *xcku025-ffva1156*. The benchmarks were chosen to range in size and FPGA resource component usage. Specifically, it was important to have benchmarks that used LUTs, FFs, BRAMs, DSPs, IOBs, and other FPGA components. For the sake of brevity, most benchmarks will not be listed here, but are available in [5]. The remainder of this section details the VDI experiments and their results, using representative benchmarks as needed.

A. RapidSmith2 Design Verification

The `VivadoConsole` is a Java class included in the RapidSmith2 distribution that is capable of creating a new instance of Vivado, sending Tcl commands to the instance, and reading back the results of the command. Using this utility, unit tests (written in the JUnit5 framework) have been created that automatically verify the correctness of a design imported from a RSCP. Figure 6 demonstrates how the `VivadoConsole` is used for these tests.

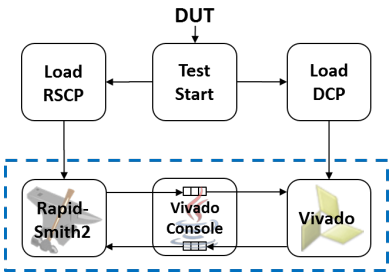

Fig. 6: RapidSmith2 Testing Flow

As the figure shows, the first step in the testing process is to load the same design into both RapidSmith2 (through RSCPs) and Vivado (through DCPs, Vivado's native checkpoint format) in parallel. Information about the Vivado design is extracted through Tcl commands sent through the `VivadoConsole`, and compared to the RapidSmith2 representation. For example, the Tcl command *[get_bel -of "myCell"]* will return the name of the BEL object that "myCell" is placed on in Vivado. This value can be compared against where RapidSmith2 thinks "myCell" is placed, and an error thrown if there is a mismatch. All logical and physical aspects of a design are tested in a similar manner.

Once the design is verified in RapidSmith2, it is exported to a TCP and loaded back into Vivado. This is a crucial step in the testing process because, as referenced in section V, several netlist modifications are performed when a TCP is generated from RapidSmith2. In Vivado, if all cells are placed and all nets are routed, then design import is declared successful. All 21 benchmarks passed these verification tests.

B. Hardware Tests

The testing process described above verifies that Vivado designs can be exported and imported without error. However, it does not test if the designs are still *functionally equivalent* and implement the same digital circuit. To test this, a set of simple FPGA designs were used. Sample applications include a VGA controller, SRAM controller, and image frame buffer (a complete list is given in [5]). Each design was first implemented and run on a Digilent Nexys4 DDR evaluation board with an Artix7 FPGA. Once the design worked on the evaluation board, it was exported from Vivado, imported into RapidSmith2, and directly exported from RapidSmith2 with no modifications. The resulting TCP was then imported back into Vivado for bitstream generation, and loaded onto the evaluation board again. If the circuit still worked as expected, the new bitstream was declared functionally equivalent to the original bitstream. All designs that were tested in this manner still executed correctly after being passed through RapidSmith2. Hardware testing is a promising first step in verifying VDI, but future work will create a more complete verification by using simulation models and test benches.

C. Case Study: Series7 Simulated Annealing Placer

To demonstrate VDI, a Series7 simulated annealing site-level placer has been implemented in RapidSmith2. The purpose of this case study is three-fold.

1) Provide an example CAD tool that operates on Vivado designs through VDI.

2) Show that VDI contains useful and sufficient information for external frameworks. When doing a site-level placement, groups of cells can generally be placed onto more than one site type. For example, a group of cells targeting a `SLICEL` site can also target a `SLICEM` site. In this case, the site types are said to be *compatible*. It is important for a site-level placer to understand which site types are compatible to create a complete list of candidate site placements for a group of cells. One aspect of VDI is to export information about compatible sites, which can be used in external tools.

3) Demonstrate the productivity advantages of using VDI with an external framework. Writing a complex simulated-annealing placer would be significantly more challenging using Vivado's Tcl API. Not only does Tcl lack higher-level programming constructs, but the placement algorithm itself would need to handle many of the challenges introduced in section III. VDI, in conjunction with RapidSmith2, abstracts much of the unnecessary detail away from the designer, and allows them to focus on the actual algorithm. Because of these abstractions, it took a student researcher about a week to create and test an initial version of the placer. It is also important to note that a placer written in Java is much faster.

After placing a design in RapidSmith2, the design is re-imported into Vivado for routing and bitstream generation. Table I shows the results of the RapidSmith2 placer vs. Vivado's default placer for a variety of designs. Figure 7 shows placement results for a Leon-3 soft processor. Source code for the placer is available in the RapidSmith2 distribution repository under the `examples.placerDemo` package.

TABLE I: Placer Results

Benchmark Name	Cell Count	Routed In Vivado?	RS2 Max Path Length	Vivado Max Path Length
SSD	59	✓	14.304ns	14.047ns
UART TX	143	✓	7.124ns	7.183ns
FIR Filter	238	✓	4.977ns	5.220ns
DiffEq	592	✓	16.335ns	17.031ns
CORDIC	2420	✓	6.211ns	5.554ns
Leon-3 [16]	12391	✓	13.105ns	8.95ns

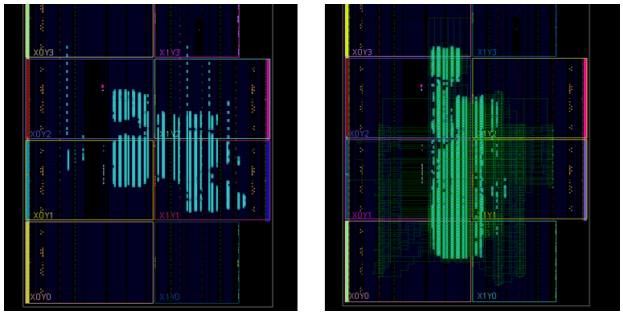

Fig. 7: Leon-3 soft processor placed by Vivado's placer (left) and RapidSmith2's simulated annealing placer (right)

D. Performance

Table II shows the import and export times of VDI for three representative benchmarks of varying sizes. As the table shows, Vivado import and export times are slow, and do not scale well. For the Leon-3 and Sha TMR designs, routing contributes to about 90% of the total export time and 95% of the total import time. One hypothesis for this behavior is that Vivado performs design rule checks for the entire design after each physical constraint is imported. This means that after each wire is assigned to a net, the entire routing structure is checked for illegal configurations. Future work will focus on improving routing import and export times for large designs. In many use cases, however, the design import/export time is not critical to the final application. The more important part is exporting and importing functionally correct designs. The initial version of VDI focuses on correctness, without concern for long import/export times.

TABLE II: Vivado Design Interface Performance[1]

Benchmark Name	Cell Count	Net Count	Slice Utilization	Export Time	Import Time
CORDIC	2420	3362	1.91%	36.66s	34.09s
Leon-3	12391	13461	15.16%	310.13s	528.88s
Sha TMR	68376	68451	84.63%	1728.66s	8110.79s

[1]Measurements were recorded on a Windows 7 64-bit workstation with a Core i7-860 processor, 8GB of DDR3 RAM and 1TB 7200RPM SATA hard disk.

VII. Conclusion

This paper presents the Vivado Design Interface, a utility to extract device and design information out of the Vivado design suite. It is meant to serve as an XDL-alternative, and has been demonstrated with the RapidSmith2 CAD tool framework. As far as we know, this work is the first successful attempt to provide an open-source tool-flow to export designs from Vivado, manipulate them in external CAD tool frameworks, and re-import an equivalent representation back into Vivado. We hope it will provide researchers a framework to explore novel CAD tools for next-generation Xilinx FPGAs.

References

[1] V. Betz and J. Rose, "VPR: A New Packing, Placement and Routing Tool for FPGA Research," in *Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications*, ser. FPL '97. London, UK: Springer-Verlag, 1997, pp. 213–222.

[2] J. Rose, J. Luu, C. W. Yu, O. Densmore, J. Goeders, A. Somerville, K. B. Kent, P. Jamieson, and J. Anderson, "The VTR Project: Architecture and CAD for FPGAs from Verilog to Routing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 77–86. [Online]. Available: http://doi.acm.org/10.1145/2145694.2145708

[3] C. Wolf and M. Lasser, "Project icestorm," http://www.clifford.at/icestorm/.

[4] T. Haroldsen, B. Nelson, and B. Hutchings, "Rapidsmith 2: A framework for bel-level cad exploration on xilinx fpgas," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 66–69.

[5] T. Townsend, "Vivado Design Interface: Enabling CAD-Tool Design for Next Generation Xilinx FPGA Devices ," Master's thesis, Brigham Young University, 2017. [Online]. Available: http://scholarsarchive.byu.edu/etd/6492

[6] C. Lavin, M. Padilla, J. Lamprecht, P. Lundrigan, B. Nelson, and B. Hutchings, "RapidSmith: Do-It-Yourself CAD Tools for Xilinx FPGAs," in *Proceedings of the 2011 21st International Conference on Field Programmable Logic and Applications*, ser. FPL '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 349–355. [Online]. Available: http://dx.doi.org/10.1109/FPL.2011.69

[7] N. Steiner, A. Wood, H. Shojaei, J. Couch, P. Athanas, and M. French, "Torc: towards an open-source tool flow," in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 41–44. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950425

[8] A. Sohanghpurwala, P. Athanas, T. Frangieh, and A. Wood, "OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, 2011, pp. 228–235.

[9] A. Otero, E. de la Torre, and T. Riesgo, "Dreams: A tool for the design of dynamically reconfigurable embedded and modular systems," in *Reconfigurable Computing and FPGAs (ReConFig), 2012 International Conference on*. IEEE, 2012, pp. 1–8.

[10] C. Lavin, B. Nelson, and B. Hutchings, "Impact of hard macro size on fpga clock rate and place/route time," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–6.

[11] A. Love, W. Zha, and P. Athanas, "In pursuit of instant gratification for fpga design," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–8.

[12] A. Das, S. Venkataraman, and A. Kumar, "Improving autonomous soft-error tolerance of fpga through lut configuration bit manipulation," in *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*. IEEE, 2013, pp. 1–8.

[13] M. Wirthlin, J. Jensen, A. Wilson, W. Howes, S.-J. Wen, and R. Wong, "Placement of repair circuits for in-field fpga repair," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2013, pp. 115–124.

[14] E. Hung, F. Eslami, and S. Wilton, "Escaping the Academic Sandbox: Realizing VPR Circuits on Xilinx Devices," in *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on*, April 2013, pp. 45–52.

[15] B. White and B. Nelson, "Tincr A Custom CAD Tool Framework for Vivado," in *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*. IEEE, Dec. 2014.

[16] C. Gaisler, "Leon3," http://www.gaisler.com/index.php/products/processors/leon3.