MS Technical Paper: Placement Algorithms for Heterogeneous FPGAs

Brian B Cheng
Rutgers University Department of Electrical and Computer Engineering

1 Keywords

- FPGA, EDA, Placement, Simulated Annealing, Optimization, RapidWright

2 Abstract

fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.
fdsafdsafdsa.

3 Introduction

Field-Programmable Gate Arrays (FPGAs) have witnessed rapid growth in capacity and versatility, driving significant advances in computer-aided design (CAD) and electronic design automation (EDA) methodologies. Since the early-to-mid 2000s, the stagnation of single-processor performance relative to the rapid increase in integrated circuit sizes has led to a design productivity gap, where the computational effort for designing complex chips continues to rise. FPGA CAD flows mainly encompass synthesis, placement, and routing; all of which are HP-hard problems, of which placement is one of the most time-consuming processes. Inefficienct placement strategy not only extends design times from hours to days, thereby elevating cost and reducing engineering productivity, but also limits the broader adoption of FPGAs by software engineers who expect compile times akin to those of conventional software compilers like `gcc`.

For these reasons, FPGA placement remains a critical research effort even today. In this paper, we study and implement established placement methods. To do this, we use the RapidWright API, which is a semi-open-source research effort from AMD/Xilinx that enables custom solutions to FPGA design implementations and design tools that are not offered by their industry-standard FPGA environment, Vivado. We implement multiple variations of simulated annealing placers for Xilinx's 7-series FPGAs, with an emphasis on minimizing total wirelength while mitigating runtime. Our implementation is organized into three consecutive substages. The **prepacking** stage involves traversing a raw EDIF netlist to identify recurring cell patterns—such as CARRY chains, DSP cascades, and LUT-FF pairs—that are critical for efficient mapping and legalization. In the subsequent **packing** stage, these identified patterns, along with any remaining loose cells, are consolidated into SiteInst objects that encapsulate the FPGA's discrete resource constraints and architectural nuances. Finally, the **placement** stage employs a simulated annealing (SA) algorithm to optimally assign SiteInst objects to physical sites, aiming to minimize total wirelength while adhering to the constraints of the 7-series architecture.

Simulated annealing iteratively swaps placement objects guided by a cost function that decides which swaps should be accepted or rejected. Hill climbing is permitted by occasionally accepting moves that increase cost, in hope that such swaps may later lead to a better final solution. SA remains a popular approach in FPGA placement research due to its simplicity and robustness in handling the discrete architectural constraints of FPGA devices. While SA yields surprisingly good results given relatively simple rules, it is ultimately a heuristic and stochastic approach that explores the vast placement space by making random moves. Most of these moves will be rejected, meaning that SA must run many iterations, usually hundreds to thousands, to arrive at a desirable solution.

In the ASIC domain, where placers must handle designs with millions of cells, the SA approach has largely been abandoned in favor of analytical techniques, owing to SA's runtime and poor scalability. Modern FPGA placers have also followed suit, as new legalization strategies allow FPGA placers to leverage traditionally ASIC placement algorithms and adapt them to the discrete constraints of FPGA architectures. While this paper does not present a working analytical placer, it will explore ways to build upon our existing infrastructure (prepacker and packer) to replace SA with AP.

4 A Brief History on FPGA Architecture

Before any work can begin on an FPGA placer, it is necessary to understand both the objects being placed and the medium in which they are placed.

Here we will briefly outline the evolution of configurable logic architecture as well as the FPGA design flow. However, readers who are already familiar with

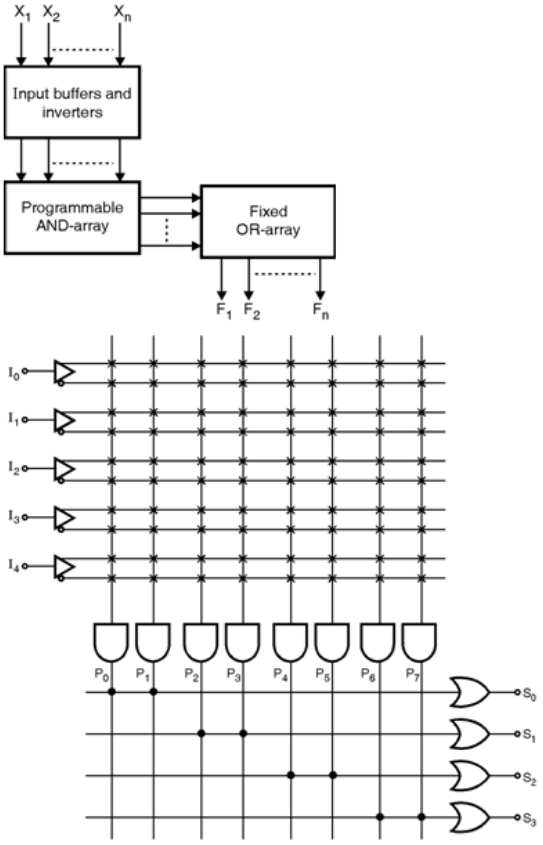

Figure 1: PAL architecture with 5 inputs, 8 programmable AND gates and 4 fixed OR gates

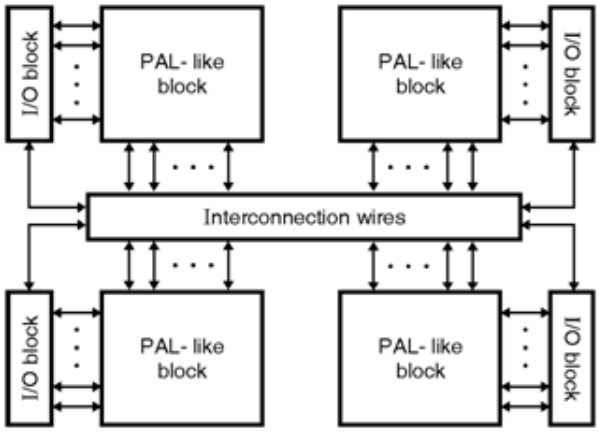

Figure 2: CPLD architecture with 4 CLBs (PAL-like blocks)

basic FPGA architecture and design flow may skip to the next section on Xilinx 7-Series Architecture. Configurable logic devices have undergone significant evolution over the past four decades. The journey began with the Programmable Logic Array (PLA) in the early 1970s. The PLA implemented output logic using a programmable-OR and programmable-AND plane that formed a sum-of-products equation for each output through programmable fuses. Around the same time, the Programmable Array Logic (PAL) was introduced. The PAL simplified the PLA by fixing the OR gates, resulting in a fixed-OR, programmable-AND design, which sacrificed some logic flexibility to simplify its manufacture. Figure 1 shows one such PLA architecture.

Later in the same decade came the Complex Programmable Logic Device (CPLD), which took the form of an array of Configurable Logic Blocks (CLBs). These CLBs were typically modified PAL blocks that included the PAL itself along with macrocells such as flip-flops, multiplexers, and tri-state buffers. The CPLD functioned as an array of PALs connected by a central programmable switch matrix and could be programmed using a hardware description language (HDL) like VHDL. Figure 2 shows one such CPLD architecture.

The mid-1980s saw the introduction of homogeneous FPGAs, which were built as a grid of CLBs. Rather than using a central programmable switch matrix as in CPLDs, FPGAs adopted an island style architecture in which each CLB is surrounded on all sides by programmable routing resources, as shown in Figure 3. The first commercially viable FPGA, produced by Xilinx in 1984, featured 16 CLBs arranged in a 4x4 grid. As FPGA technology

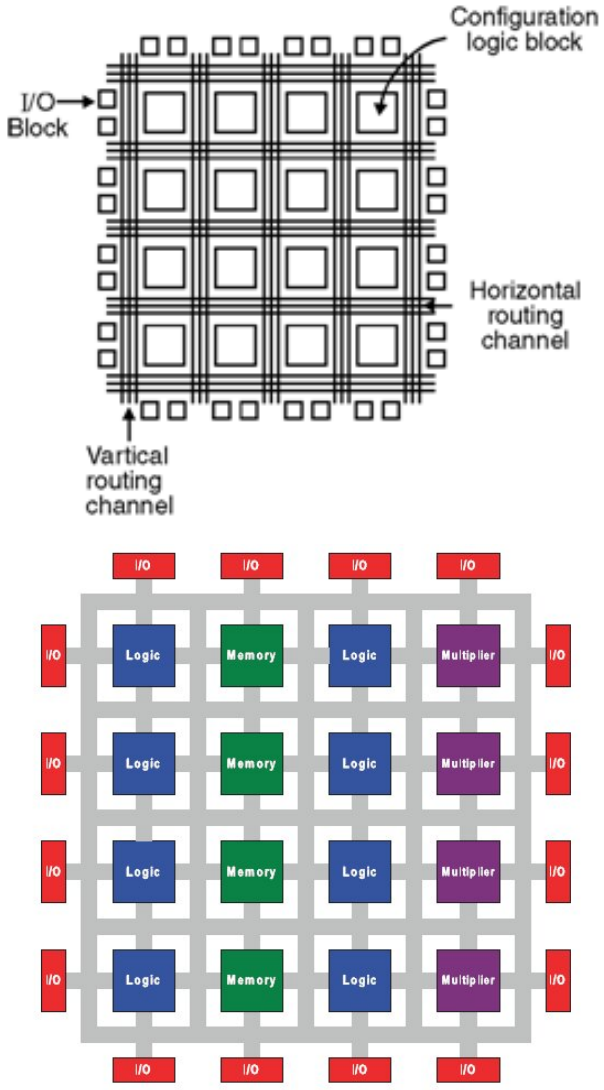

Figure 3: **Top**: A homogeneous island-style FPGA architecture with 16 CLBs in a grid. **Bottom**: A heterogeneous island-style FPGA with a mix of CLBs and macrocells.

advanced, CLBs were redesigned to use lookup tables (LUTs) instead of PAL arrays for greater logic density. The capacity of an FPGA was often measured by how many logical elements or CLBs it offered, which grew from hundreds to thousands and now to hundreds of thousands of CLBs.

This brings us to modern day FPGA architectures. To meet the needs of increasingly complex designs, FPGA vendors introduced heterogeneous FPGAs. In these devices, hard macros such as Block RAM (BRAM) and Digital Signal Processing (DSP) slices are integrated into the programmable logic fabric along with CLBs, like shown in Figure 3. This design enables the direct instantiation of common subsystems like memories and multipliers, without having to recreate them from scratch using CLBs. Major vendors such as Xilinx and Altera now employ heterogeneous island-style architectures in their devices. As designs become increasingly large and complex, FPGAs meet the demand by becoming increasingly heterogenous by incorporating a wider variety of hard macros into the fabric.

5 Xilinx 7-Series Architecture

In this paper we will be working with the Xilinx 7-Series devices which follows the heterogenous island-style architecture. The 7-Series architecture first released in 2010 and was superceded by the Ultrascale architecture which made several improvements over the 7-Series. Although the latest FPGAs follow the Ultrascale architecture, this paper focuses on the 7-Series FPGAs due to their greater accessibility to beginner users and compatibility with open source tooling.

Figure 4 illustrates the architectural hierarchy of a 7-Series Xilinx FPGA. At its most basic level, an FPGA is a vast physical array of replicated atomic components known as Basic Elements of Logic (BELs). These BELs, which include LUTs, FFs, BRAMs, DSPs, and programmable interconnects, form the fundamental building blocks for custom digital circuit implementations on FPGAs.

The Xilinx architectural hierarchy organizes these BELs into increasingly abstract structures. **BELs** are packaged into **Sites**, which are then embedded into **Tiles**, which are subsequently consolidated into **Clock Regions**. In high-end Xilinx devices, Clock Regions may be further consolidated into **Super Logic Regions** (SLRs). However, for the scope of this paper, we focus on Xilinx devices that have only a single SLR.

Note that in the Xilinx architectures, "CLBs" refer to CLB Tiles, each containing two "SLICE" Sites, with each SLICE Site containing 8 LUTs, 8 Flip Flop, and 1 CARRY BEL. These SLICEs facilitate the core reprogrammability of the FPGA.

6 The FPGA Design Flow and Toolchain

Figure 5 illustrates a typical FPGA design and verification workflow, transforming a high-level digital design into a configuration file that programs the physical FPGA.

The process begins with **design entry**, where an engineer describes the desired functionality using a hardware description language (HDL) such as Verilog or VHDL. Engineers may use various coding styles - behavioral, structural, dataflow, or a mix of all three - to capture the logic and behavior of the design while abstracting away the intricacies of the physical hardware architecture.

Next, the user runs the **synthesis** tool, which parses the raw HDL source files to infer logical behavior, perform optimizations, and instantiate primitive cells. The result is a detailed netlist, typically output as a low-level Verilog or Electronic Design Interchange Format (EDIF) file, where the design is represented purely in structural HDL. The final stage is often referred to as "technology mapping", as the vocabulary or library of primitive cells must be specific to the FPGA architecture and consequently the FPGA manufacturer.

After synthesis, the **placement** tool assigns each logic element in the netlist a specific location within the FPGA's fabric. Depending on the placement strategy, this stage can include substages of prepacking, packing, global placement, legalization, and detailed placement. Our placer will only involve prepacking, packing, and detailed placement for simplicity. The objective is to place the logical cell instances of the netlist onto physical BELs on the device in a way that minimizes delay, satisfies design constraints, and simplifies routing.

Following placement, the **routing** tool connects the placed elements by assigning physical wiring resources (switchboxes, programmable interconnects) to the nets in the netlist. The router determines their paths through the programmable interconnect network while attempting to meet timing and performance constraints.

Placement and routing are generally referred jointly as the **implementation** stage, as both stages are interdependent and must adhere to the constraints of the same device. State of the art (SOTA) tools will perform routing-aware placement for greater optimization.

The final stage is the **bitstream** generation, where a vendor-specific programming tool produces a configuration file or bitstream that programs the physical FPGA device. The bitstream specifies the state of every configurable element—logic blocks, routing switches, and hard macros—ensuring the FPGA implements the user's design.

Parallel to this entire design flow are simulators for which the user can write testbenches to verify their design at different stages of implementation. Xilinx's Vivado, for example, provides functional simulation, post-implementation simulation, and post-implementation timing simulation.

This integrated toolchain, whether provided by FPGA vendors or open source communities, facilitates the transformation from an abstract HDL description to a functioning hardware configuration.

7 RapidWright API

We use the RapidWright API to implement a custom placer for Xilinx 7-Series FPGAs. RapidWright is an open source Java framework that enables netlist implementation manipulation for modern Xilinx FPGAs.

Figure 6 shows how the API allows us to take design

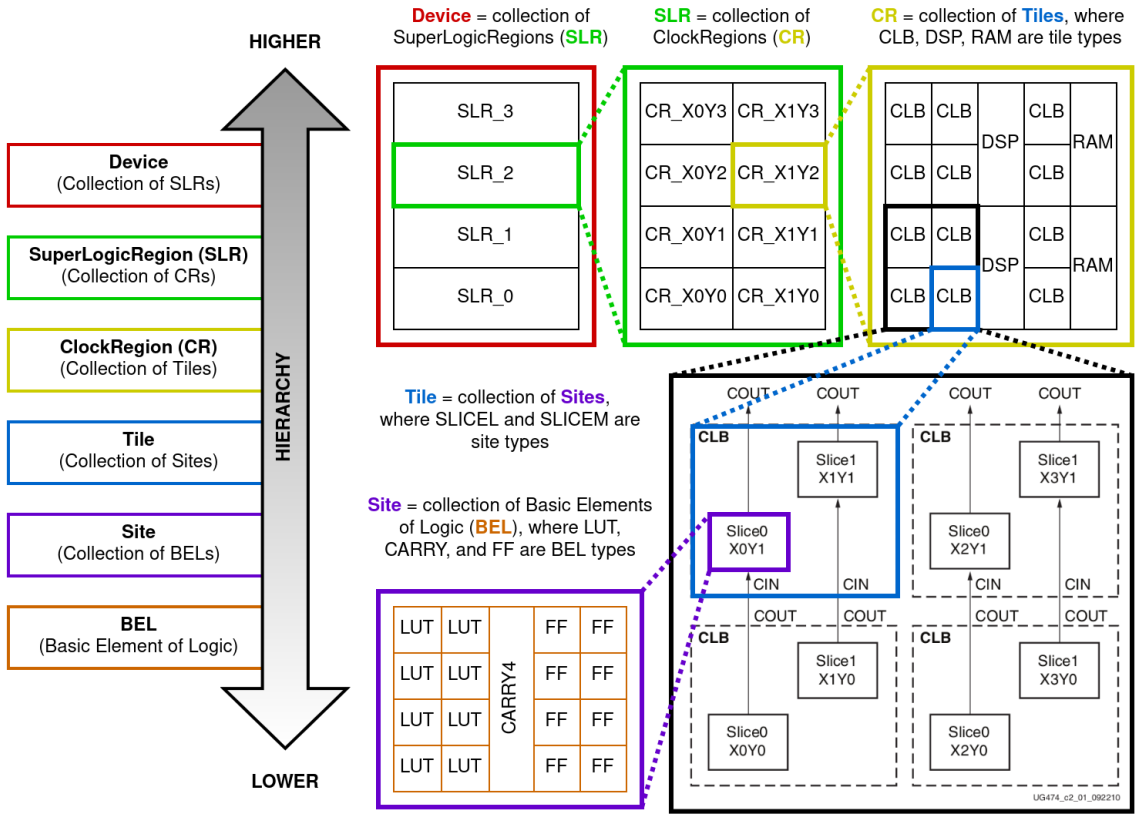

Figure 4: Architecture Hierarchy of a Xilinx FPGA

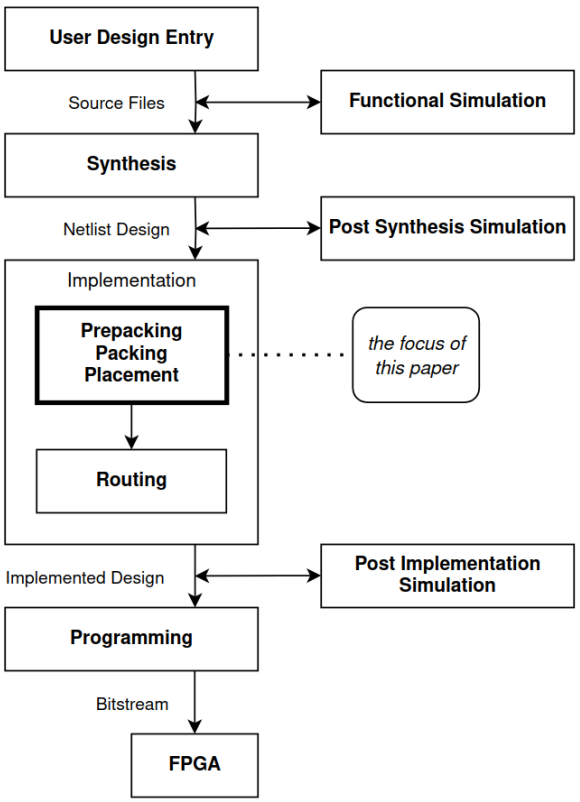

Figure 5: A typical FPGA design and verification workflow

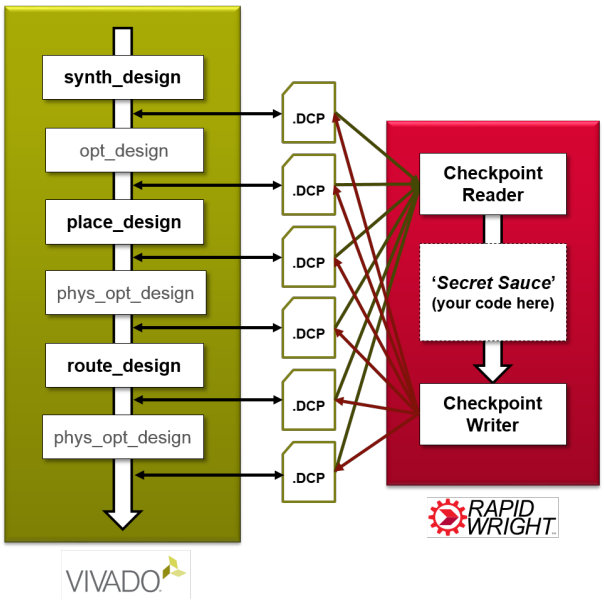

Figure 6: Showing how RapidWright can take a design checkpoint from any stage of the design flow, allow user customization, and reintegrate it back into the design flow.

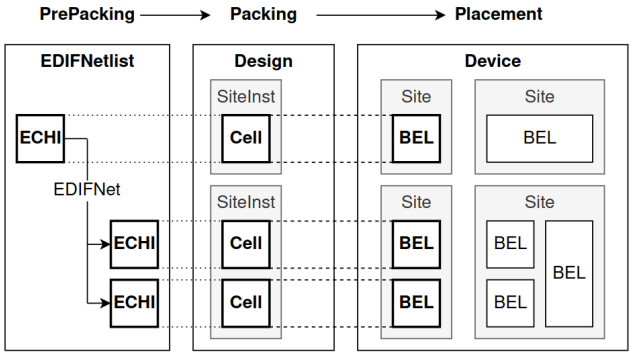

Figure 7: The placement flow illustrating PrePacking into Packing into Placement for a logical netlist with 3 EDIF cells. EHCI is short for EDIFHierCellInst.

checkpoints in the form of .dcp files generated at any stage of the design flow, allow the user to make custom optimizations, and reintegrate it back into the design flow. This allows the user to substitute any stage in the default FPGA design flow offered by Vivado with a custom user-defined process to meet complex design constraints. There are three packages of objects that are most relevant to building our placer: the **edif** package, **design** package, and **device** package. Figure **??** shows the EDIF netlist objects and device routing objects as shown in the Vivado netlist viewer and device viewer. Figure 7 illustrates the flow between these packages in relation to our PrePacking, Packing, and Placement flow.

RapidWright reads in a post-synthesis design checkpoint and allows us to traverse the netlist's logical EDIFCells through various function calls. For each EDIFCell object in the netlist, we create a Cell design object, which we then place onto physical device BEL object. CellPins and Nets are automatically generated from EDIFHierPortInst and EDIFHierNet upon the Cell's creation from an EDIFHierCellInst.

These three steps correspond to our placement substages of PrePacking, Packing, and Placement, respectively.

For a complete understanding of the RapidWright packages, please refer to the API Overview `https://www.rapidwright.io/docs/RapidWright_Overview.html` and the API reference.

8 Placement Implementation

With a basic understanding of FPGA architecture, design placement, and RapidWright, we have all the necessary pieces to implement our SA placer. Here we outline in detail each substage of our implementation: PrePacking, Packing, and Placement.

8.1 PrePacking

In the prepacking stage, we traverse the raw post-synthesis EDIF Netlist to identify common recurring Cell patterns. We do this because there are certain desirable cell configurations that should be clustered together to exploit some proximity characteristics of the FPGA architecture. For example, in Figure **??** Doing so will innately improve wirelength minimization. There are also some cell configurations that must necessarily be placed in certain ways in relation to each other to ensure legality within the architecture constraints.

For example, in many HDL designs, LUTs and FFs often come in pairs, as is the case in Figure **??**

Furthermore, certain cell patterns are specific to certain cell types. Therefore, we must first traverse the entire EDIF netlist and identify all unique cell types and group them together via a `HashMap<String, List<EDIFHierCellInst>>`, where key String is the name cell type and value `List<EDIFHierCellInst>` is the list of cells of that type. The resulting lists in the hashmap are mutually exclusive.

Cell types can include `CARRY4`, `LUT1-6`, `FF`, `DSP48E1`, `RAMB18E1`, and others. We specifically look for CARRY4 chains, DSP48E1 cascades, and LUT-FF pairs as these patterns are common to nearly all FPGA designs. We will describe each cell pattern in detail.

8.2 CARRY Chains

Figure **??** shows an example of a CARRY4 cell while figure **??** shows an example of a CARRY4 chain of size 6.

As the name implies, CARRY chains are chains of CARRY cells. Very often an HDL design will implement many adders, counters, subtractors, or comparators, all operations based on binary addition. They are so ubiquitous that every that in the 7-Series architecture, every CLB comes embedded with a CARRY4 BEL, a 4-bit carry-lookahead (CLA) adder that can be chained together across CLBs to implement wider adders.

(By now Need to have introduced SLICEL / SLICEM Sites and differentiated them from "CLBs", which are referred to as CLBL or CLBM Tiles) (rip some figures from the 7-series CLB user guide)

These CARRY4 cells, if chained together, must be placed vertically across CLBs, as each CLB is connected vertically by a Carry In (CI) to Carry Out (COUT) wire between them. The only way to route the carry net is through this wire, thus why these CLBs must be placed consecutively vertically. This allows the facilitation of fast adders on the device.

The CARRY4 We identify CARRY4 chains by taking the CARRY4 entry of the HashMap. We follow the below pseudocode

Carry chain cells 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 with raw LUTs would results in an inefficient use of resources and performance would suffer. In this way, the CARRY4 cell is like a macrocell.

On the device, they must necessarily be arranged consecutively vertically, from bottom to top. Figure **??** shows an example of a CARRY4 EDIFCell chain in an EDIFNetlist. Notice how there are many different types of cells connected to the CARRY4 Cells. The raw netlist does not tell us the presence of any carry chains or other cell clusterings. We must manually traverse the netlist to identify all of the carry chains by accessing each carry cell, accessing its Ports, accessing the net on the port, accessing all of the other ports on the net, and checking if those ports belong to another carry cell. We continue to

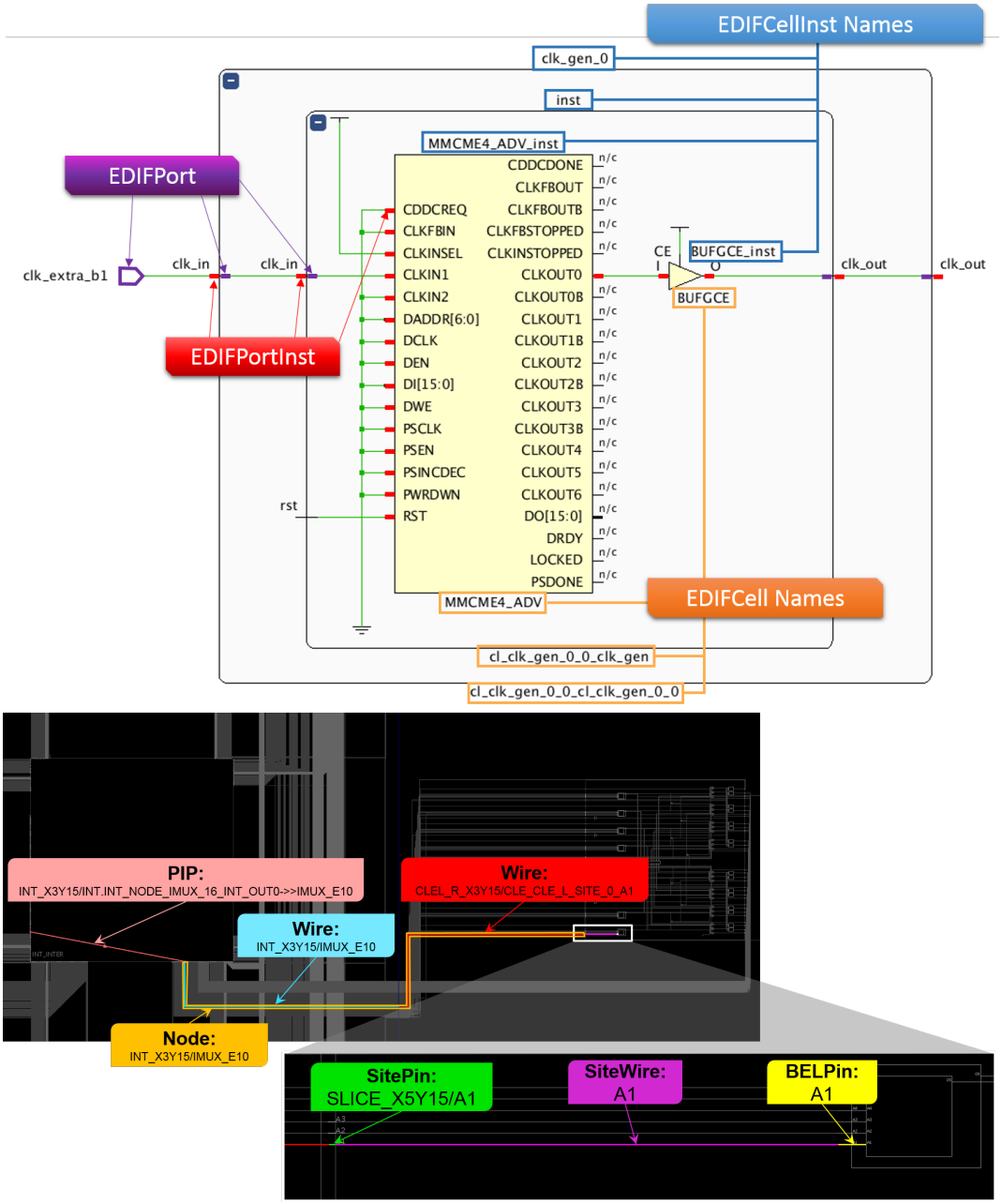

Figure 8: Examples of edif netlist objects (top) and routing objects (bottom) for a Xilinx FPGA

traverse the carry chain until the final carry cell is not connected to another carry cell. We must traverse in both directions. After we reach the tail of the current chain, we traverse in the opposite direction (CIN vs COUT) to find the carry chain anchor. We then remove all the cells in this chain from the set of all carry cells. We select the next carry cell in the set of all carry cells and repeat the process, until there are no more remaining carry cells in the set of all carry cells.

Shown in figure **??**

To find CARRY chains, we take the CARRY4 group and follow the below pseudocode. Each SLICE Site contains one CARRY4 BEL. Each CLB Tile contains two SLICE Sites. CARRY chains span vertically across multiple SLICE Sites/Tiles. (Show picture)

8.3 DSP Cascades

Each DSP Site contains one DSP BEL. Each DSP Tile contains two DSP Sites. DSP chains can span vertically across multiple DSP Sites/Tiles.

8.4 LUT-FF Pairs

Identify unique CE-SR net pairs. All FF cells in a Site must share the same CE and SR nets.

8.5 Packing

In the packing stage, we take the identified cell clusters and package them into SiteInst Design objects which target the Device Site objects.

8.6 CLB Sites

Can support LUT-FF pairs, loose LUTs, loose FFs, CARRY chains. Each SLICE has 8 ”lanes” of LUT-FFs. 4 LUT5s and 4 LUT6s. 8 FFs. For SLICEMs, LUT6s can be configured as shallow 32-bit LUTRAMs or ”RAMS32”.

Figure 9: The data classes populated at each substage: PrepackedDesign, PackedDesign, and PlacedDesign.

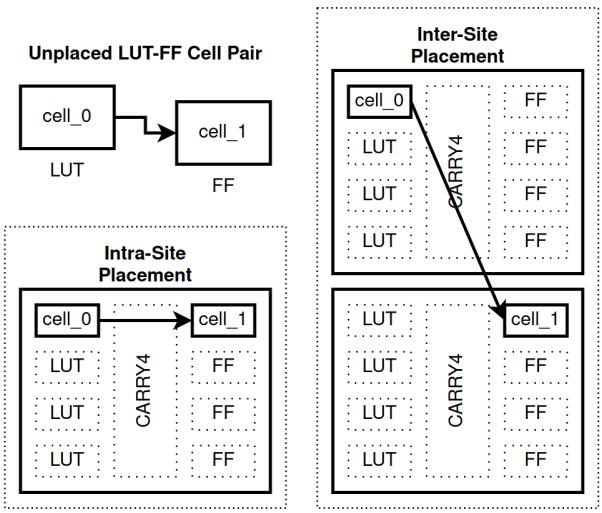

Figure 10: **Upper Left**: A LUT-FF cell pair in the netlist. **Upper Right**: The LUT-FF pair placed within the same Site. **Bottom**: The LUT-FF pair placed across different Sites.

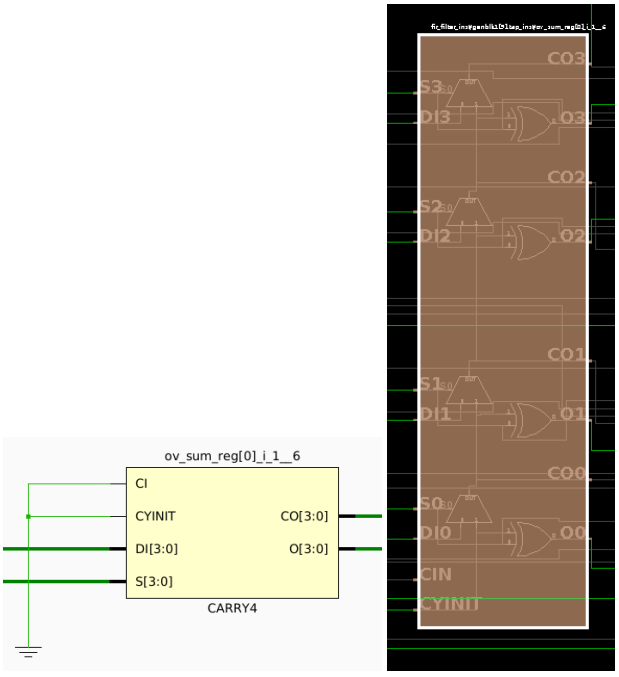

Figure 11: **Left**: A CARRY4 EDIFCell as seen in the Vivado netlist viewer. **Right**: The corresponding CARRY4 Cell as seen in the device viewer.

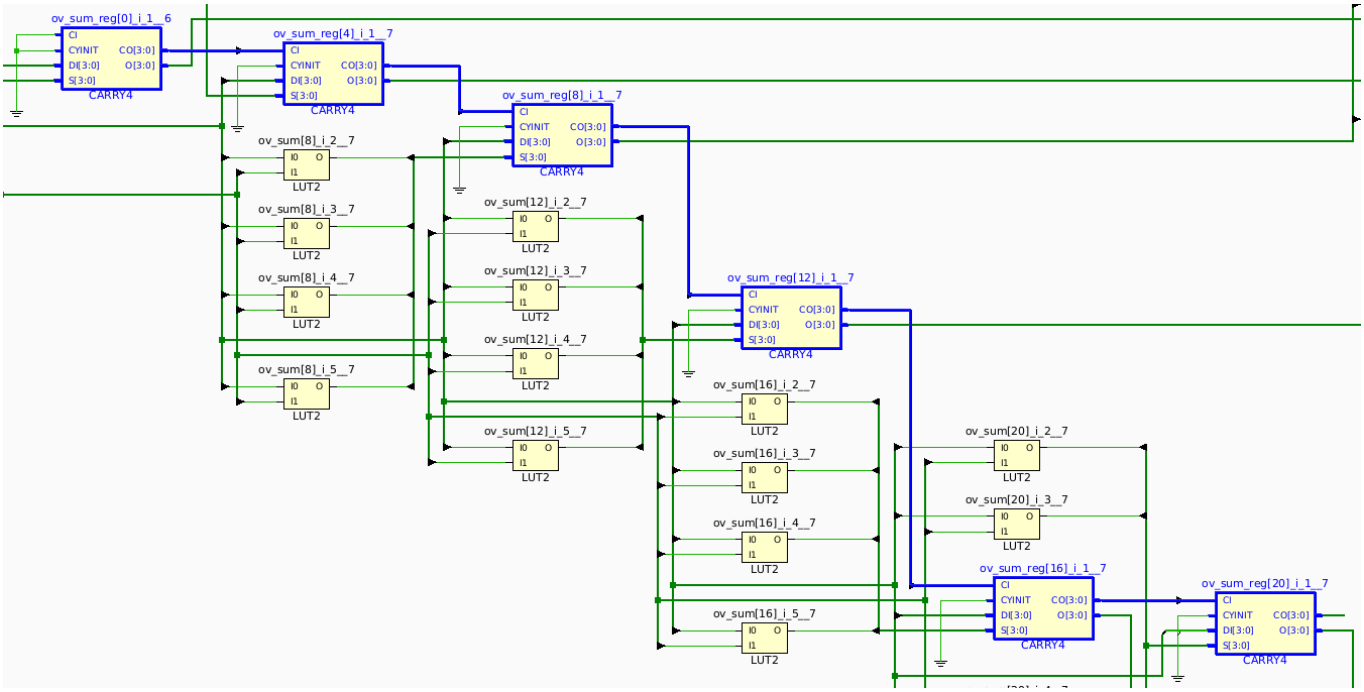

Figure 12: Example of a CARRY4 chain of length 6 as seen in the Vivado netlist viewer.

8.7 Placement: Simulated Annealing

Bookkeeping: keep track of Sites occupied by single SiteInsts and Sites occupied by SiteInst chains. Create BEL "fields": CLB, DSP, CLB and DSP Chains, RAMs.

8.8 Detailed Placement

9 Placement Results

- Random Site Selection:
- Midpoint Site Selection:
- Hybrid: Random Site selection until progress stagnates, then use Midpoint Site selection. Randomly choose betwee Random and Midpoint Site selection each iteration.
- Cooling Schedule: Greedy. Geometric with modifiable alpha and intial temperature.

10 Future Work

- Different cooling schedules: linear cooling, logarithmic cooling, piecewise cooling.
- Support for more macros: buffer cells, FIFO, clock generators, etc..
- Support Ultrascale architecture.
- Better packing scheme. Currently pseudorandom packing.
- Inter-Site BEL swapping.
- BEL-centric placement over Site-centric placement.
- Analytical placement. Global placement, legalization, detailed placement.

11 Conclusion

-

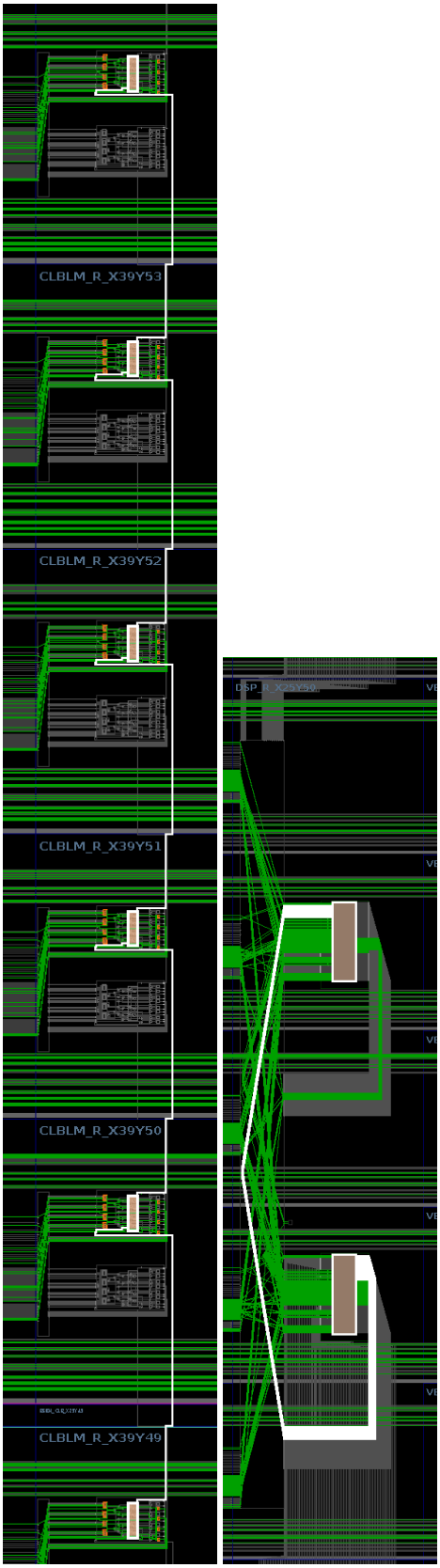

Figure 13: **Left**: A Carry Chain of length 6 spanning 6 Sites across 6 Tiles. **Right**: A DSP cascade of length 2 spanning 2 Sites across 1 Tile.