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.
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.

The paper first begins by elaborating on general FPGA architecture and then specifically the Xilinx 7-Series architecture. Then, the paper will elaborate on the FPGA design flow, then the role that the RapidWright API plays in the design flow. We explain in detail each of these concepts for a broader audience as they provide much needed context for FPGA placement algorithms as a concept. However, readers who are already familiar with these concepts can skip directly to the RapidWright API section 7 or to the Simulated Annealing section 8.

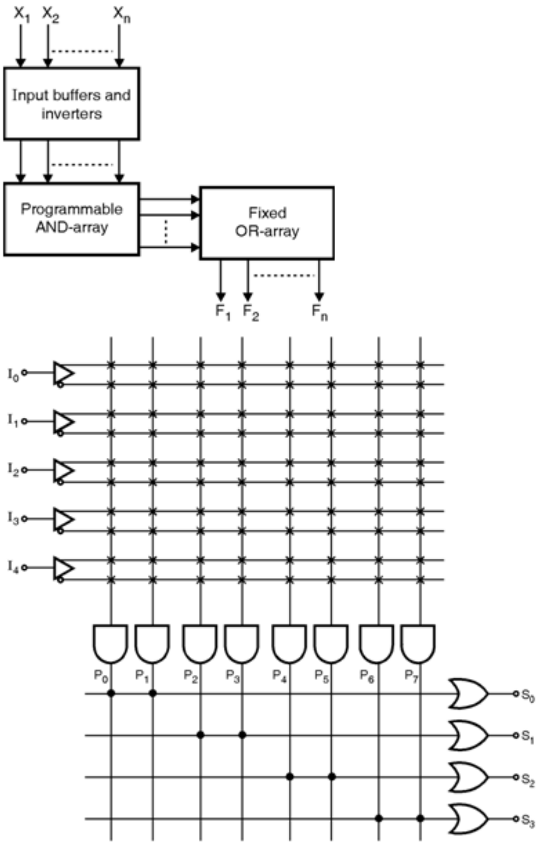

Figure 1: PAL architecture with 5 inputs, 8 programmable AND gates and 4 fixed OR gates

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. Configurable logic devices have undergone significant evolution over the past four decades. We will briefly review the evolution of configurable logic architecture starting in the 1970s and quickly work our way up to modern day FPGA architecture.

PLA: 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.

CPLD: 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.

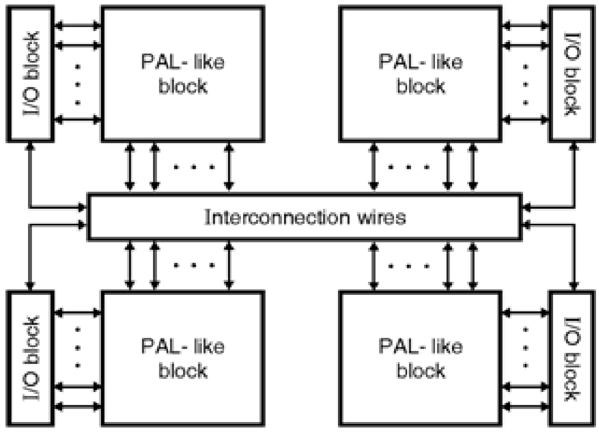

Figure 2: CPLD architecture with 4 CLBs (PAL-like blocks)

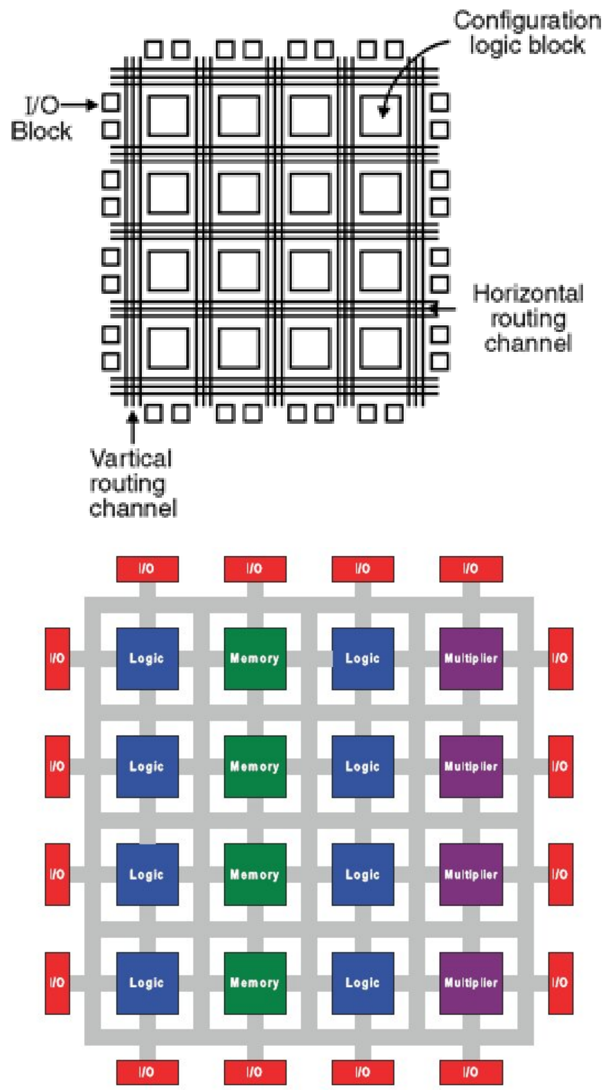

Figure 3: **Top**: A homogeneous island-style FPGA architecture with 16 CLBs in a grid. **Bottom**: A heterogeneous island-style FPGA with a mix of CLBs and macrocells.

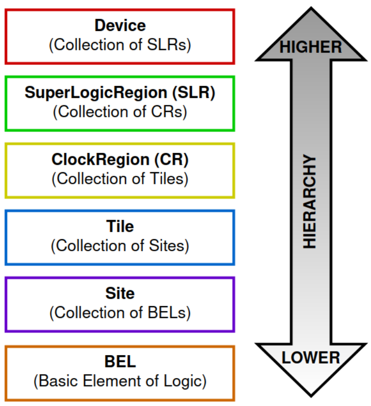

Figure 4: Overview of Architecture Hierarchy of a Xilinx FPGA

Homogeneous FPGA: 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 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.

Heterogenous FPGA: 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, incorporating a wider variety of hard macros into the fabric.

5 Xilinx 7-Series Architecture

The Xilinx 7-Series devices, first introduced in 2010, follow a heterogeneous island-style architecture as discussed previously. Although the 7-Series was later superseded in 2013 by the UltraScale architecture, the 7-Series remains highly relevant due to its accessibility, wide availability, and compatibility with open-source tooling. Representative sub-families include Artix-7, Kintex-7, Virtex-7, and Zynq-7000, each designed with different performance and cost trade-offs but all follow the core 7-Series architecture.

Figure 5 illustrates a high-level view of the hierarchical organization of a 7-Series FPGA. At its lowest level, the device consists of a large array of atomic components called *Basic Elements of Logic* (BELs). These BELs encompass look-up tables (LUTs), flip-flops (FFs), block RAMs (BRAMs), DSP slices (DSPs), and the configurable interconnect fabric. They constitute the fundamental building blocks for implementing digital circuits on the FPGA.

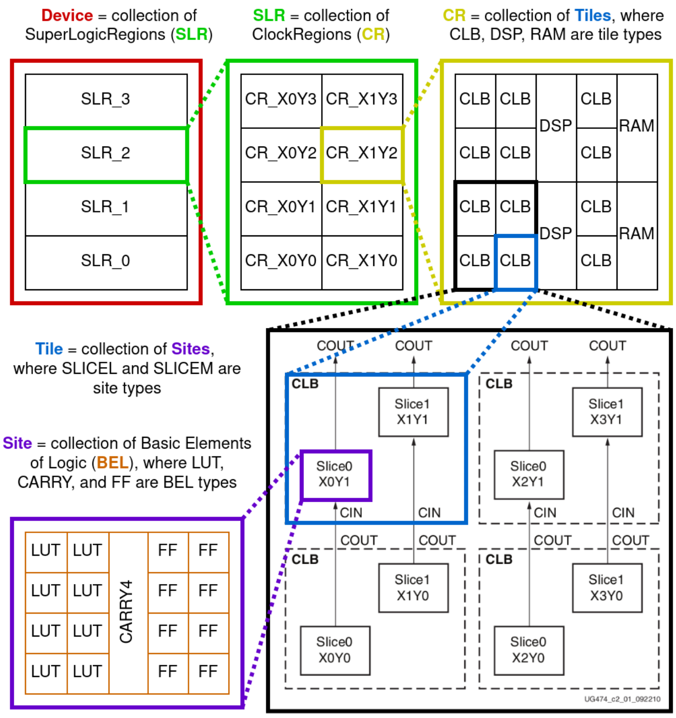

Figure 5: Detailed Architecture Hierarchy of a Xilinx FPGA

To manage this complexity, Xilinx organizes these BELs into incrementally abstract structures. First, **BELs** are grouped into **Sites**. Each Site is embedded into a **Tile**, and Tiles are further arranged into **Clock Regions**. In some high-density devices, multiple Clock Regions may be consolidated into one or more **Super Logic Regions** (SLRs). However, for the scope of this paper, we focus on Xilinx 7-Series devices with only a single SLR.

In the 7-Series architecture, the term *CLB* (Configurable Logic Block) refers to a *CLB Tile* that contains two *SLICE* Sites. Each SLICE Site has a set of BELs including eight LUTs, eight FFs, and one *CARRY4* adder. The BELs in these SLICEs facilitate the bulk of the general programmability of the FPGA fabric. We will explain in detail the motivation and function of these BELs.

LUTs Combinational logic is universal to all HDL designs. As the their name suggests, Look-Up Tables (LUTs) facilitate combinational logic by acting as tiny asynchronously-accessed ROMs whose contents are fixed when the FPGA is programmed. For any boolean function, the synthesizer precalculates the output to every possible input combination and stores the resulting truth table into a LUT's static memory. The inputs are then essentially treated as an address space that maps to a data value space in an asynchronous ROM. No explicit logic gates like NAND or XNOR are synthesized, contrary to what newcomers might expect from a "Field Programmable Gate Array". In the 7-Series devices, one LUT can facilitate any 6-input boolean function, or two 5-input functions, as long as they share the same input signals. The LUT can also host two independent boolean functions of up to 3 inputs each, even when the inputs

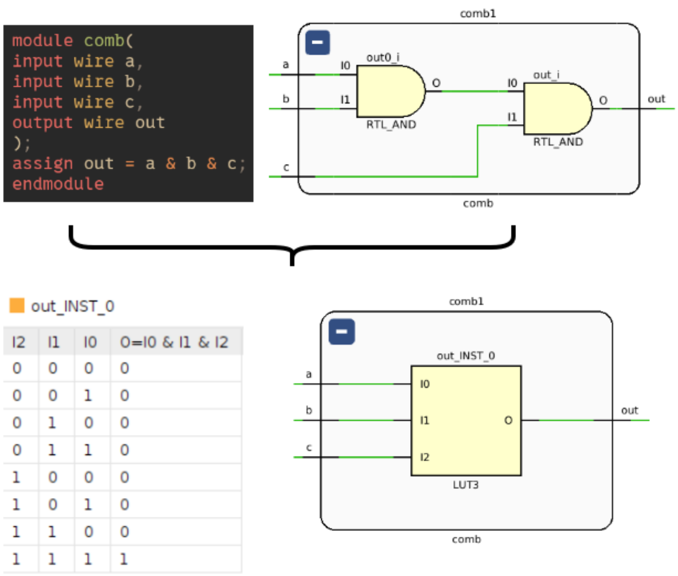

Figure 6: LUT synthesis from user design

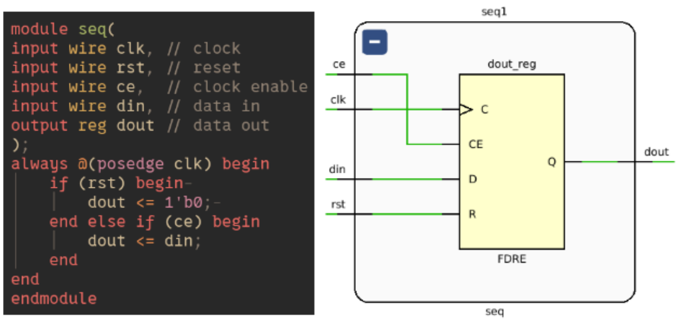

Figure 7: FF synthesis from user design

are not shared. Functions requiring more than six unique inputs are decomposed across multiple cascaded LUTs. Figure 6 shows an example of where a LUT is typically synthesized in a design entry.

FFs FFs are synthesized to facilitate synchronous event-driven signal assignment. For most FPGA users, in Verilog, this essentially means assignments wrapped in `always @(posedge clk)` statements. Figure 7 shows an example of where a flip flop (FF) is typically synthesized. The cell primitive FDRE is a type of FF and belongs to a family of D Flip Flops (DFFs) with Clock Enable (CE).

- **FDCE:** DFF with CE and Asynchronous Clear
- **FDPE:** DFF with CE and Asynchronous Preset
- **FDSE:** DFF with CE and Synchronous Set
- **FDRE:** DFF with CE and Synchronous Reset

In a typical HDL design, the vast majority of FFs will be synthesized as FDREs. A FF BEL may also be used to implement latches, however, since latches are generally known to be bad practice in FPGA design, we will not consider latch synthesis for this paper.

LUT-FF Pairs

CARRY Very often an HDL design will implement many adders, counters, subtractors, or comparators, all of which are based on binary addition. They are so ubiquitous that every that in the 7-Series architecture, every SLICE features a *CARRY4* BEL – a 4-bit carry-lookahead (CLA) adder. These CARRY4 blocks can be chained vertically across SLICEs for implementing wide adders efficiently.

CARRY Chains

SLICE Types. Xilinx offers two variants of SLICE Sites: **SLICEL** and **SLICEM**.

- *SLICEL (SLICE-Logic):* Each LUT in a SLICEL can implement a 6-input Boolean function (with up to two outputs under certain input-sharing conditions). During logic synthesis, larger Boolean expressions exceeding six inputs are automatically split across multiple LUTs.

- *SLICEM (SLICE-Memory):* The SLICEM include all the features of a SLICEL but its LUTs can also be configured as small distributed RAM (known as *Distributed RAM*) or as shift registers. This local memory capability supplements the larger Block RAM (BRAM) cells and is beneficial for designs requiring numerous small RAM blocks.

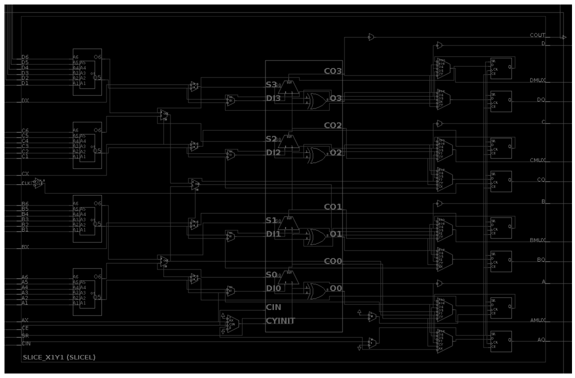

Figure 8: A SLICEL Site in the Vivado device viewer

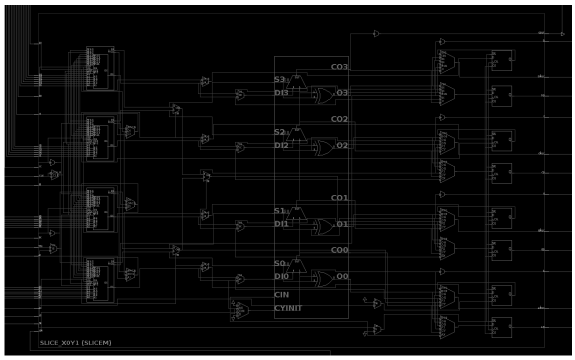

Figure 9: A SLICEM Site in the Vivado device viewer

In a typical 7-Series device, approximately 75% of the SLICE Sites are SLICELs and 25% are SLICEMs. A single CLB Tile can therefore host either two SLICELs or one SLICEL and one SLICEM.

(Explain cell patterns here)

DSPs

DSP Cascades

BRAMs

Loose Single Cells

For more in-depth details about 7-Series FPGAs, refer to the official Xilinx user guides such as:

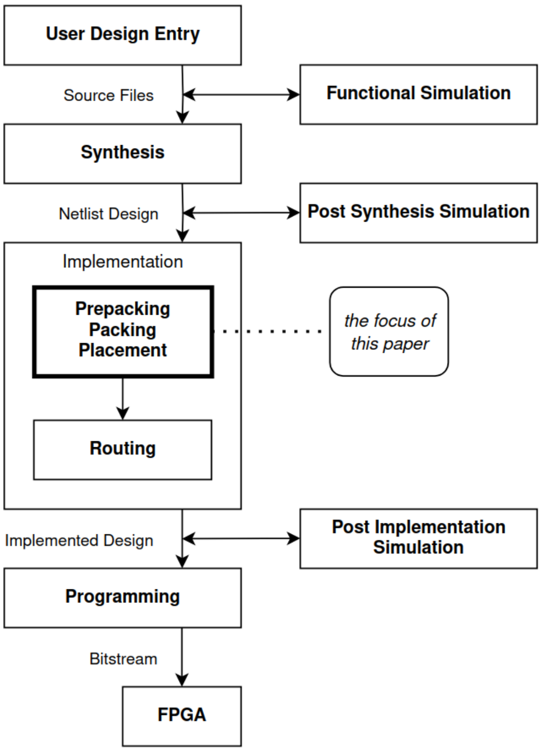

Figure 10: Typical FPGA design and verification workflow.

- *7 Series FPGAs Overview* (UG476)
- *7 Series FPGA Configurable Logic Block* (UG474)
- *7 Series Memory Resources* (UG473)
- *7 Series DSP48E1 Slice* (UG479)

This architectural context provides the necessary background for understanding how a placement algorithm should account for resource constraints and optimize performance in modern FPGA designs.

6 FPGA Design Flow and Toolchain

Modern FPGA designs require a sophisticated toolchain to bridge the gap between high-level hardware descriptions and the final bitstream used to configure the FPGA. Figure 10 illustrates a representative process that converts an abstract Hardware Description Language (HDL) design into a verified configuration file for a target device.

1. **Design Entry:** An engineer describes the intended functionality of the digital system using a hardware description language (HDL) such as Verilog or VHDL. During this phase, the coding style can vary (behavioral, structural, dataflow, etc.), but it always aims to capture high-level behavior rather than device-specific details.

2. **Synthesis:** The synthesis tool parses the HDL source, performs logical optimizations, and maps the design onto primitive cells that suit the target FPGA technology. The output is typically a *structural* netlist (e.g., EDIF or structural Verilog) which details how the design's logic is broken down into LUTs, FFs, and other vendor-specific cells.

3. **Placement and Routing (Implementation):** In *placement*, each logical cell from the synthesized netlist is assigned to a physical location on the FPGA fabric. For instance, LUTs and FFs go into specific *BELs* within the device's CLB sites, and specialized cells such as DSPs and Block RAMs must be placed in their corresponding tile types. Next, *routing* determines how signals are physically wired through the FPGA's configurable interconnect network. Modern tools often interleave these steps (e.g., fluid-placement routing or routing-aware placement) to better meet timing and area objectives.

4. **Bitstream Generation:** After a design is fully placed, routed, and timing-closed, the toolchain produces a final *bitstream* that sets the configuration of every programmable element in the FPGA. This bitstream can then be loaded onto the device, either through vendor software or via a custom programming interface.

5. **Verification:** In parallel to the design flow, simulations and testbenches validate correctness of the user's design at multiple abstraction levels. Engineers may begin with functional or behavioral simulations, then progress to post-synthesis simulations, and finally to post-implementation simulations that incorporate real routing delays. With each higher level of fidelity, computational requirements grow significantly due to increasing complexity and the need to analyze more variables over time. Ensuring correct functionality and meeting timing closure at the post-implementation stage is crucial before deploying the design to hardware. Given the importance of thorough verification, many established companies dedicate one verification engineer for every design engineer.

This workflow underscores the critical role of **placement** in bridging the netlist to a physical realization. An efficient placement algorithm can drastically reduce compile times and improve design performance, enabling broader adoption of FPGAs in application spaces that require fast design iteration.

7 RapidWright API

RapidWright is an open-source Java framework from AMD/Xilinx that provides direct access to the netlist and device databases used by vendor tools. This framework positions itself as an additional workflow column, allowing users to intercept or replace stages of the standard design flow with custom optimization stages (see Figure 11).

- **Design Checkpoints:** RapidWright leverages `.dcp` files (design checkpoints) generated at various stages of a Vivado flow. By importing a checkpoint, engineers can manipulate the netlist, placement, or routing externally, then re-export a modified checkpoint for further processing in the Vivado workflow column.

- **Key Packages:** RapidWright revolves around three primary data model packages:

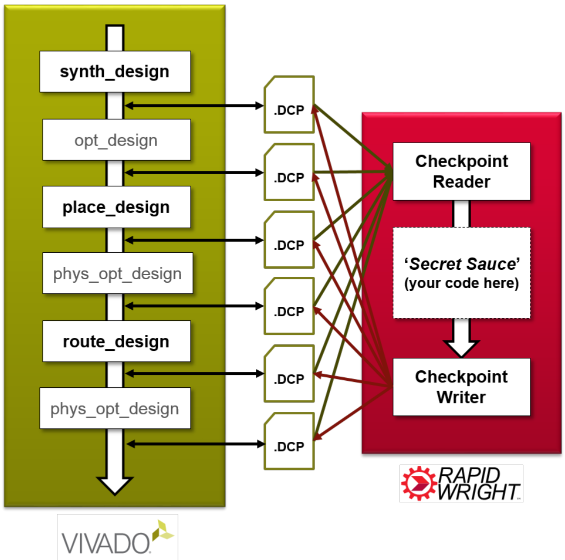

Figure 11: RapidWright workflow integrating into the default Vivado design flow.

1. *edif* – Represents the logical netlist in an abstracted EDIF-like structure.
2. *design* – Contains data structures for the physical implementation (Cells, Nets, Sites, BELs, etc.).
3. *device* – Provides a database of the target FPGA architecture (e.g., Site coordinates, Tile definitions, routing resources).

- **Interfacing with the Netlist and Device:** An engineer can query the netlist to find specific resources (LUTs, FFs, DSPs, etc.) and then map or move them onto device sites. This level of control over backend resources is necessary for research in custom placement, advanced packing techniques, or experimental routing algorithms.

By exposing these low-level internals, RapidWright allows fine-grained design transformations that go beyond the standard Vivado IDE's capabilities. Researchers can prototype new EDA strategies without needing to re-implement an entire FPGA backend from scratch, thus accelerating innovation in placement and routing methodologies.

8 Simulated Annealing

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. Shown in Figure

8.1 Prepacking

In the context of implementing a custom *Simulated Annealing* (SA) placer, **prepacking** is the first step in consolidating functionally related cells from the raw post-synthesis netlist into logical clusters that correspond more naturally to the physical FPGA structures. We traverse the raw post-synthesis netlist via RapidWright's

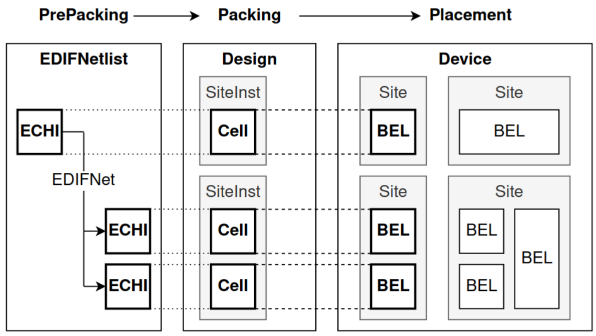

Figure 12:

`edif` packages to identify LUT-FF Pairs, CARRY4 Chains, and DSP Cascades.

LUT-FF pairs: Most synchronous designs implement finite-state machines or pipelined datapaths, mapping their combinational logic into LUTs directly feeding FFs. Clustering each LUT-FF pair under a single "prepacked object" facilitates usage of dedicated intra-SLICE connections, thereby reducing routing overhead.

CARRY4 chains: Arithmetic operations (adders, counters, comparators) often chain multiple *CARRY4* cells vertically. During prepacking, it is crucial to detect these chains so they remain grouped and placed in physically consecutive and vertical SLICEs for placement legality and optimal performance.

DSP cascades: In designs with large multiply-accumulate structures, DSP48E1 blocks can be used and cascaded along dedicated paths. Identifying such cascades early helps the placer keep them localized, minimizing high-speed interconnect usage. Like carry chains, each DSP in a DSP cascades must be placed vertically, in physically consecutive and vertical DSP48E1 Sites.

Loose or Single Cells The remaining cells Various design-specific macros (e.g., BRAM-based FIFOs, shift-register logic) also benefit from being clustered, though these may be addressed in subsequent or more advanced stages.

This strategy not only reduces the complexity of the placement search space but also helps ensure legality (e.g., shared control signals, shared carry nets). The subsequent *packing* and *placement* steps can then operate on these higher-level groupings with improved efficiency.

8.2 Packing

8.3 Placement

Up until now we have only organized the logical `Cells` into `SiteInsts`. This is where simulated annealing actually begins.

Figure 13: The data classes populated at each substage: PrepackedDesign, PackedDesign, and PlacedDesign.

o

9 OLD: Simulated Annealing

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.

9.1 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 the physical organization of the BELs and Sites of the architecture. Doing so will innately improve wire-length 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.

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.

9.2 LUT-FF Pairs

Very often, engineers model the behavior of a digital system as a collection of Finite State Machines (FSM) as shown in Figure 15. An FSM at its core is a synchronous sequential circuit comprising of a combinational element

Figure 14: The data classes populated at each substage: PrepackedDesign, PackedDesign, and PlacedDesign.

and a synchronous memory element. In the 7-Series architecture, combinational elements are synthesized as LUTs, where the truth table of the combinational function is mapped onto the LUT's memory.

Combinational functions in 7-Series FPGAs are synthesized as Look-Up Tables (LUTs). The FPGA synthesizer maps the combinational element into LUTs and the memory element onto FFs.

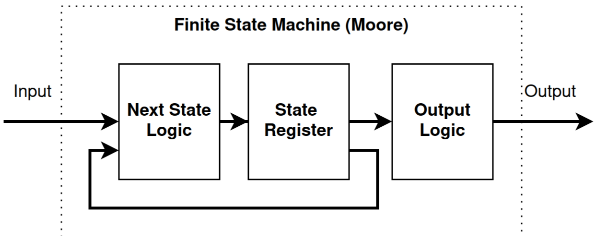

Figure 15: A Finite State Machine (Moore)

In the vast majority of HDL designs, LUTs and FFs will be synthesized in pairs, as is the case in Figure 16. Shown in Figure 17 are two possible placements for this cell net. On the right, the cells are placed across different Sites, thus the only way to route the net between the cells is through general inter-site routing. On the bottom left, the cells are placed within the same Site, taking advantage of the intra-site routing and avoiding costly inter-site routing.

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

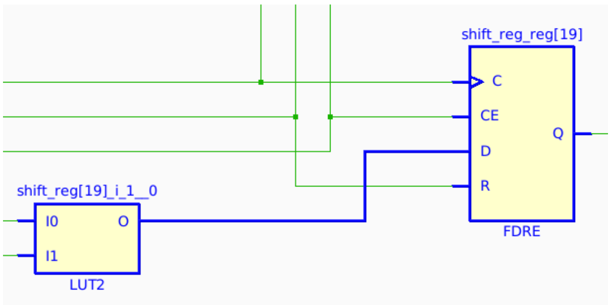

Figure 16: Example of a LUT-FF Cell Pair as shown the Vivado netlist viewer. FDRE is a type of Flip Flop.

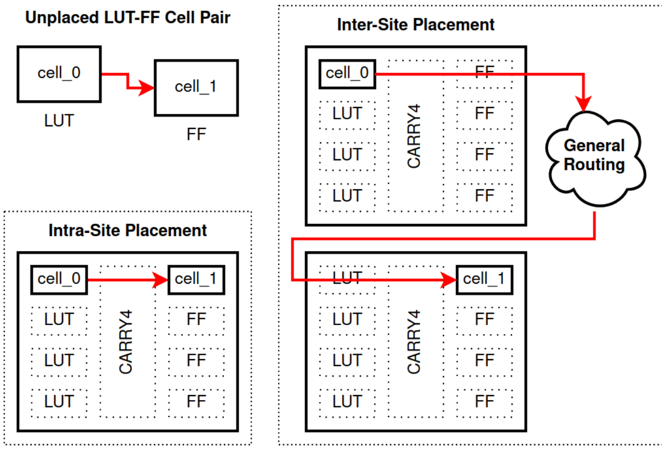

Figure 17: **Upper Left**: A LUT-FF cell pair in the netlist. **Upper Right**: The LUT-FF pair placed within the same Site. **Bottom**: The LUT-FF pair placed across different Sites.

9.3 CARRY Chains

Figure 18 shows an example of a CARRY4 cell while figure 19 shows an example of a CARRY4 chain of size 6.

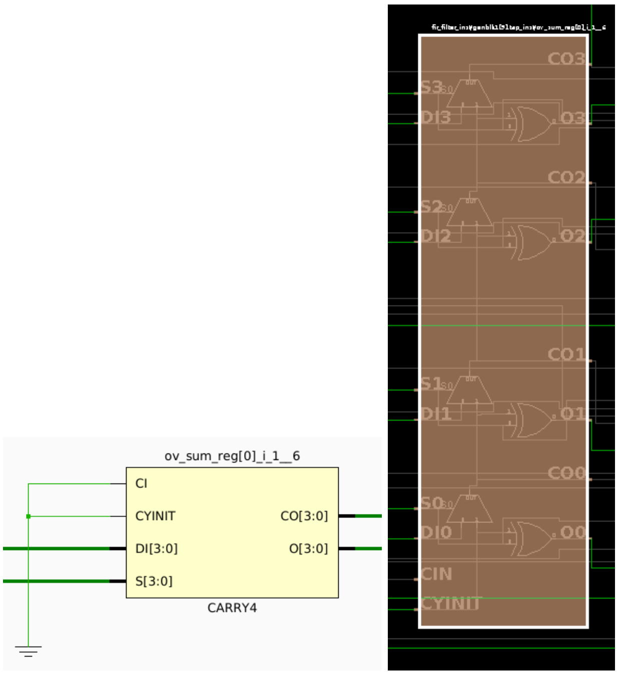

Figure 18: **Left**: A CARRY4 EDIFCell as seen in the Vivado netlist viewer. **Right**: The corresponding CARRY4 Cell as seen in the device viewer.

As the name implies, CARRY chains are chains of CARRY cells.

(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 19 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 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 18

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)

9.4 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.

9.5 Packing

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

9.6 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".

9.7 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.

9.8 Detailed Placement

10 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.

11 Future Work

- Different cooling schedules: linear cooling, logarithmic cooling, piecewise cooling.
- Support for more macros: buffer cells, FIFO, clock generators, etc..
- Support Ultrascale architecture.

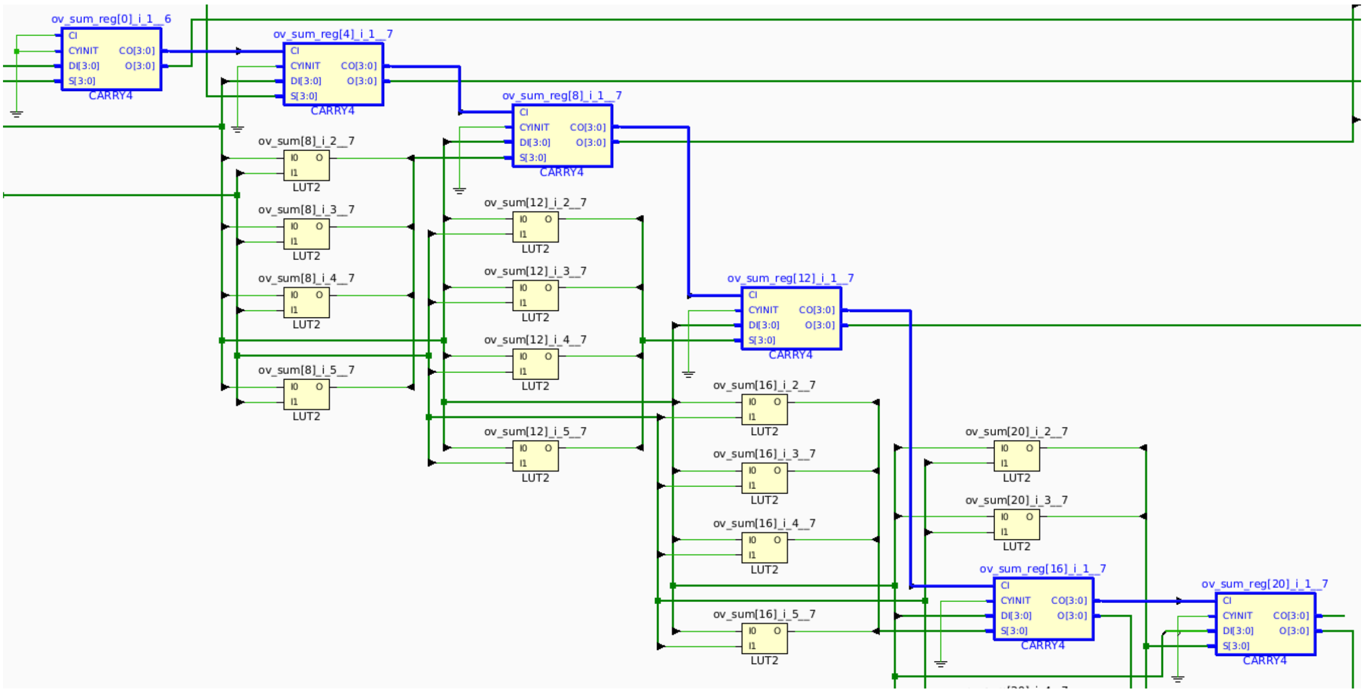

Figure 19: Example of a CARRY4 chain of length 6 as seen in the Vivado netlist viewer.

- Better packing scheme. Currently pseudorandom packing.
- Inter-Site BEL swapping.
- BEL-centric placement over Site-centric placement.
- Analytical placement. Global placement, legalization, detailed placement.

12 Conclusion

fdsfdsafdsa

13 Appendix

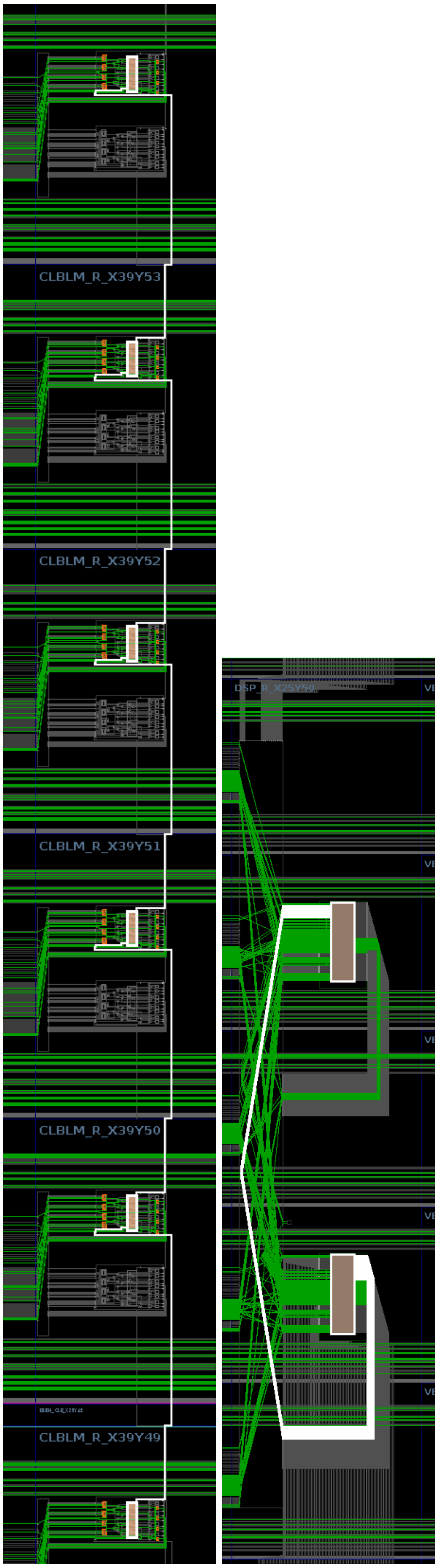

Figure 20: **Left**: A Carry Chain of length 6 spanning 6 Sites across 6 Tiles. **Right**: A DSP cascade of length 2 spanning 2 Sites across 1 Tile.

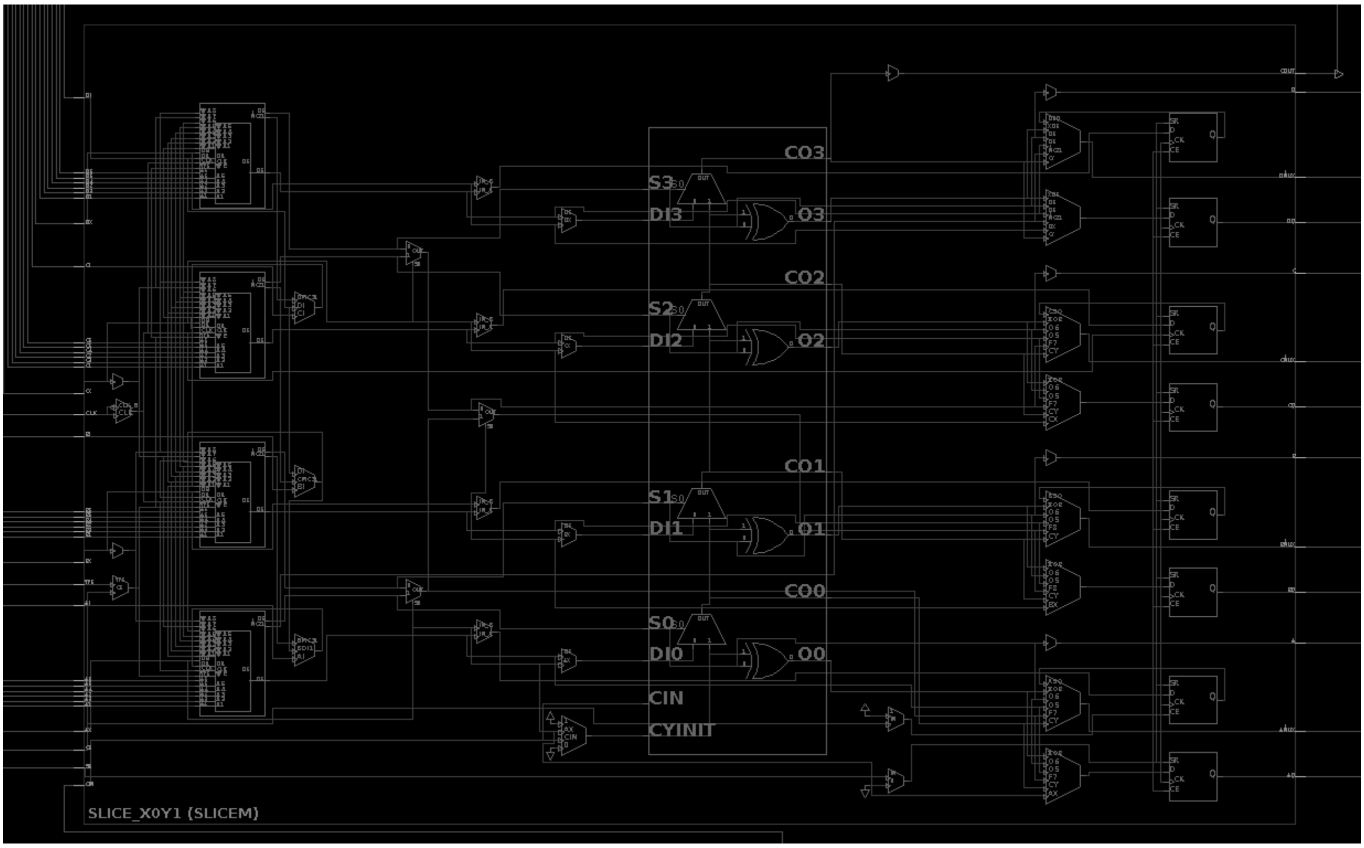

Figure 21: SLICEM Site

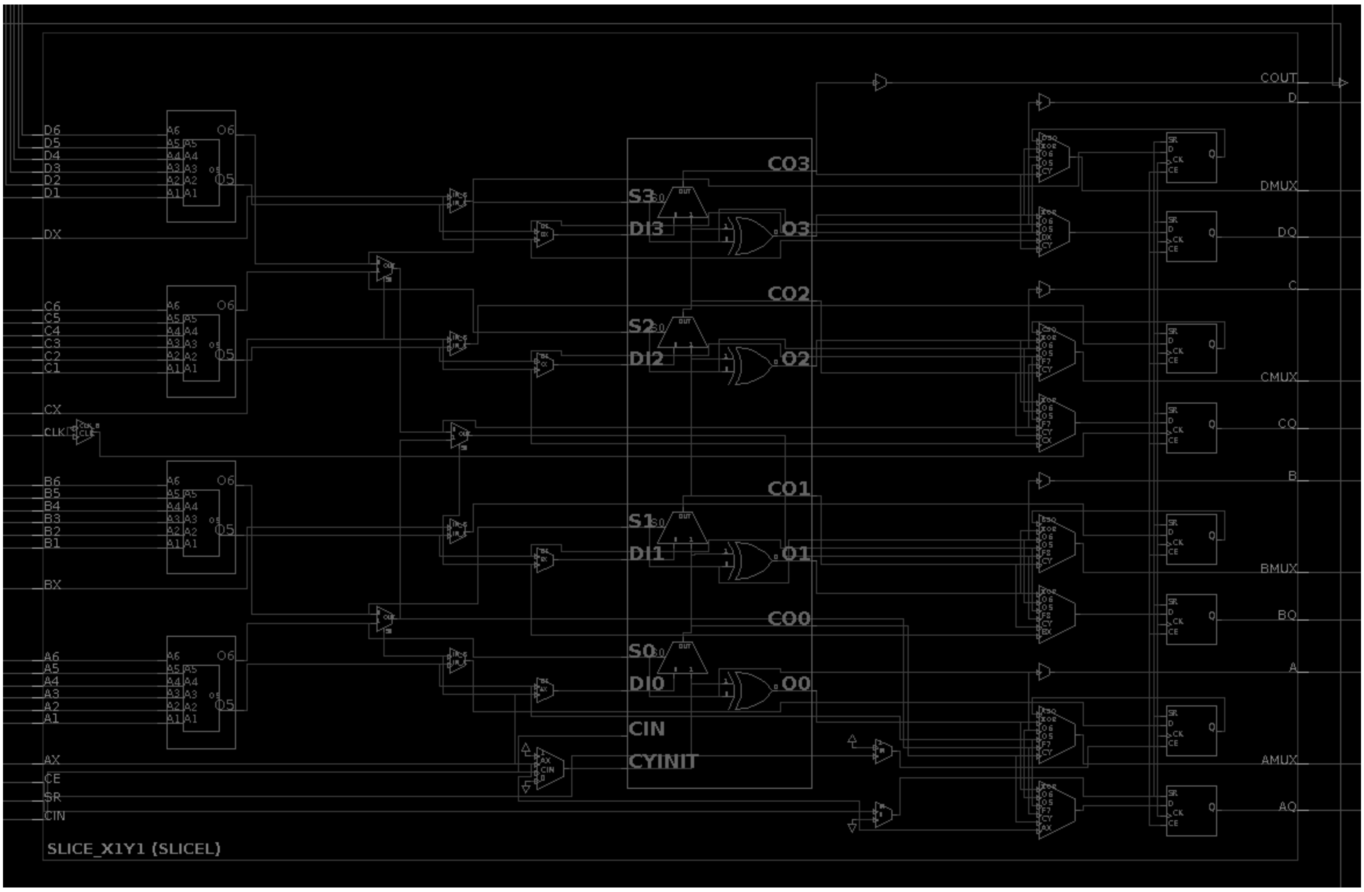

Figure 22: SLICEL Site