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 NP-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 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 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.

4 FPGA Architecture History

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 PAL 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.

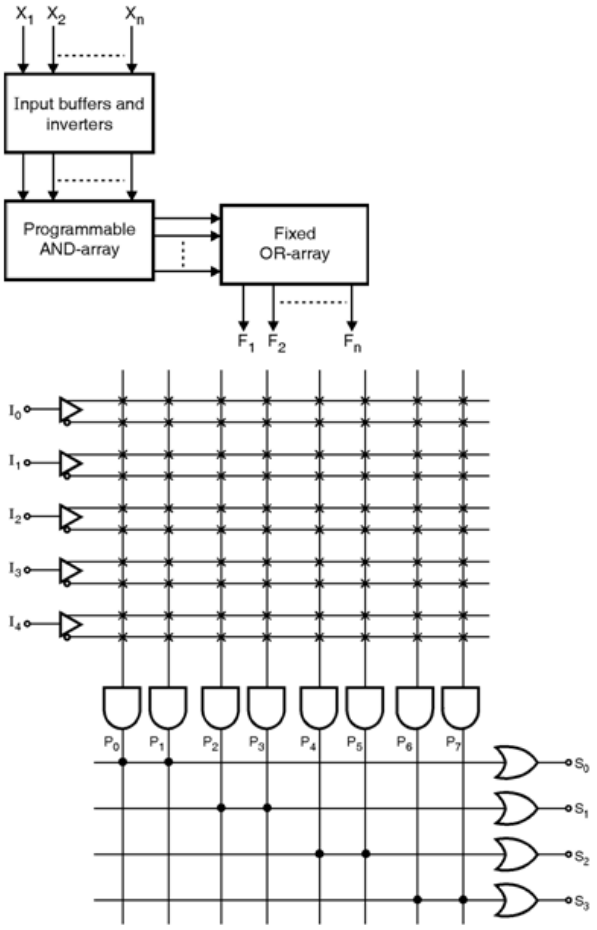


Figure 1: PAL architecture with 5 inputs, 8 programmable AND gates and 4 fixed OR gates

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 4. 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.

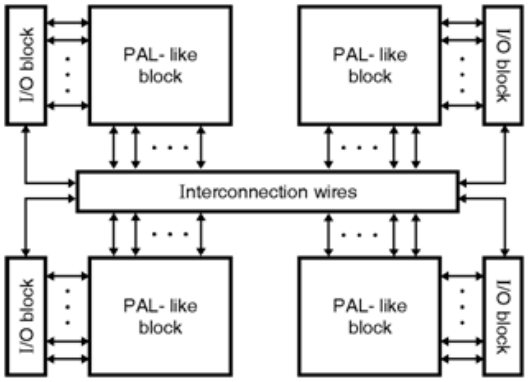


Figure 2: CPLD architecture with 4 CLBs (PAL-like blocks)

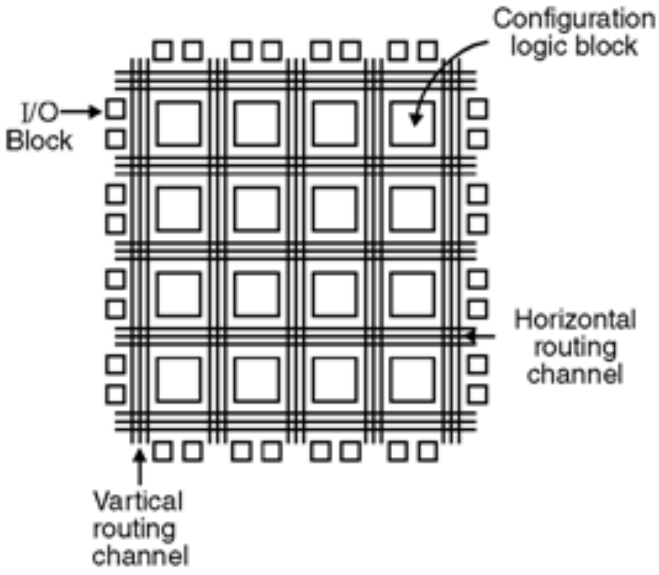


Figure 3: A homogeneous island-style FPGA architecture with 16 CLBs in a grid.

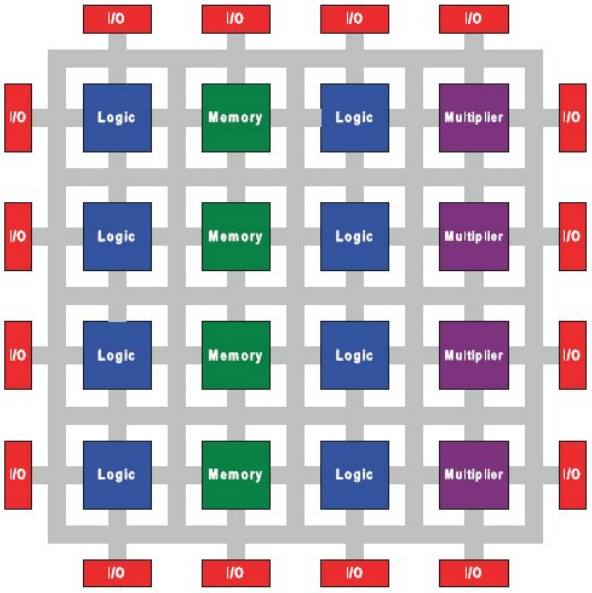


Figure 4: A heterogeneous island-style FPGA with a mix of CLBs and macrocells.

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 4. 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. Note how the Tile arrangement is columnar, where each column will have only one Tile type.

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.

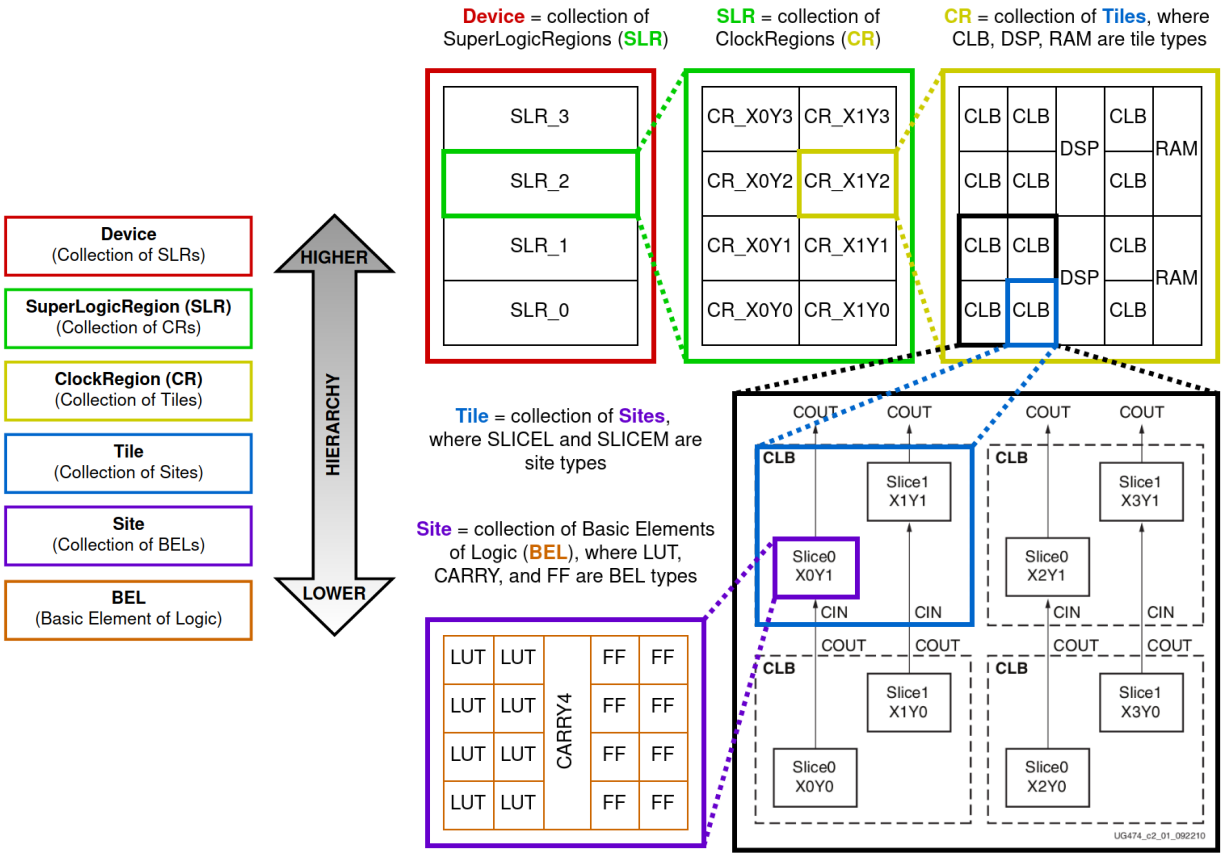


Figure 5: Architecture Hierarchy of a Xilinx FPGA

5.1 CLB SLICEs

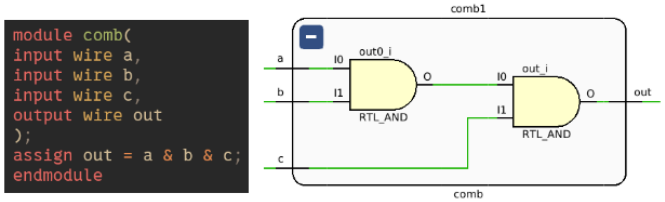
In the 7-Series architecture, the term *CLB* (Configurable Logic Block) refers to a *CLB Tile* that contains two *SLICE* Sites. Xilinx offers two variants of SLICE Sites: **SLICEL** and **SLICEM**.

- Each `SLICEL` has a set of BELs including eight `LUT`s, eight `FF`s, and one `CARRY4` adder. The `LUT` BELs in a `SLICEL` can only host `LUT` Cells.
- The `SLICEM` includes all the features of a `SLICEL` but its `LUT` BELs can host both `LUT` Cells, which are asynchronous ROM elements, or `RAM32M` Cells, which are synchronous 32-deep RAM elements. These cells are also referred to as *Distributed RAM* in the Xilinx documentation. These cells can offer an alternative to the larger, more dedicated 18K-36K `RAMB18E1` cells when RAM resources are highly utilized.

In a typical 7-Series device, approximately 75% of the SLICE Sites are `SLICEL`s and 25% are `SLICEM`s. A single CLB Tile can therefore host either two `SLICEL`s or one `SLICEL` and one `SLICEM`. To simplify the problem space, however, we will only consider `SLICEL`s for general logic and use the dedicated `RAMB18E1` cells to implement RAM elements.

The BELs in these SLICEs facilitate the bulk of the general programmability of the FPGA fabric. We will explain in detail the function and motivation behind these BELs.

LUTs Combinational logic is universal to all HDL designs. As the their name suggests, a Look-Up Table (LUT) map an input value to an output value. 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 boolean 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".



out_INST_0

I2	I1	I0	O=I0 & I1 & I2
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

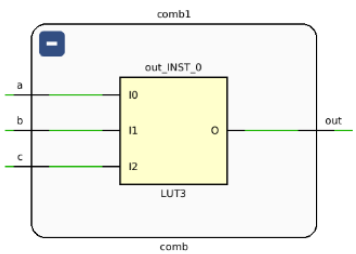


Figure 6: LUT synthesis from user design

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 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 Verilog users, this generally means signal assignments wrapped in `always @(posedge clk)` statements. Figure 7 shows an example of where a 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 with the occasional FDSE, as it is generally good practice to keep FPGA designs synchronous. A FF BEL may also host a LATCH Cell, however, since they are generally bad practice in FPGA design, we will not consider latches in this paper.

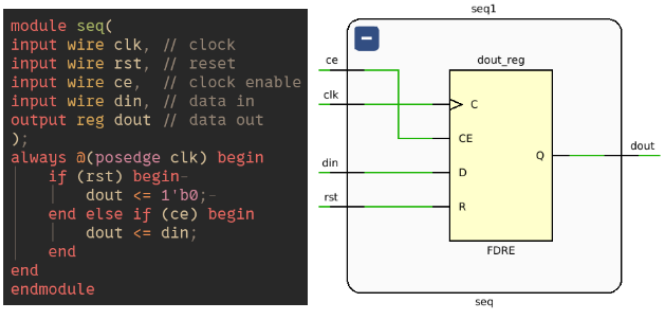


Figure 7: FF synthesis from user design

Up to eight (8) FFs can be placed within the same SLICE, but only if they all share a common Clock-Enable (CE) net and a common Set-Reset (SR) net. This is because the SLICE has only one CE pin and one SR pin to interface with general routing. The CE and SR signals from these pins are broadcast intra-Site to all FFs within.

LUT-FF Pairs FPGA designs are very often modelled as a collection of Finite State Machines (FSM) like shown in Figure 8. Many times a design will also use pipelining, either to model signal buffers or shift registers, or to split up large combinational logic blocks into time slices to meet timing constraints. These common design structures result in many consecutive sections of combinational logic feeding into a vector of registers. The synthesizer naturally synthesizes these structures as consecutive pairs of LUTs feeding into FFs as shown in Figure 9. Figure 10 shows an example of a synthesized LUT-FF Pair.

Shown in Figure 11 are two possible placements for a LUT-FF Pair on the physical device. On the right, the cells are placed across different Sites, thus the only way to route the net between the cells is through general intersite routing. On the bottom left, the cells are placed within the same Site in the same lane, taking advantage of the intra-site routing without burdening the general router with additional inter-site routing. This is an important consideration to make while minimizing wirelength during placement.

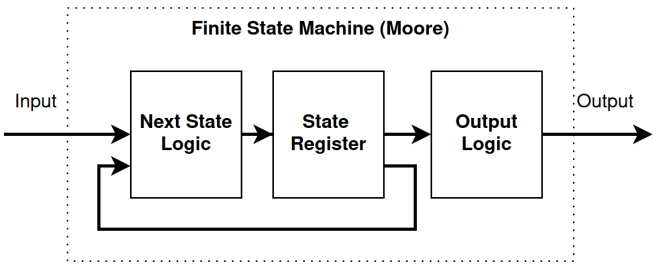


Figure 8: Finite state machine (Moore)

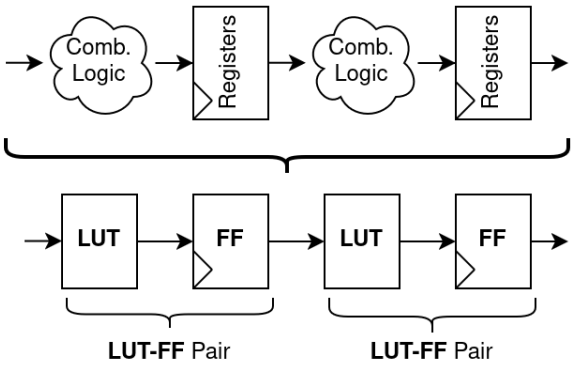


Figure 9: Pipelining synthesized as consecutive LUT-FF pairs

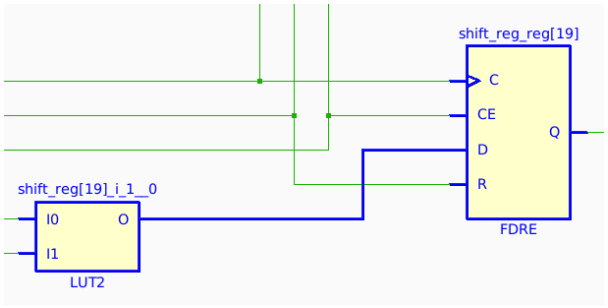


Figure 10: A synthesized LUT-FF Pair

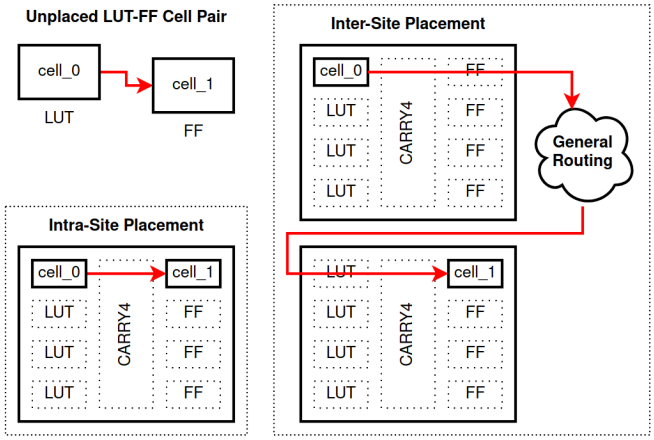


Figure 11: Intrasite vs Intersite LUT-FF Placement

.

CARRY An FPGA design will also typically 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, a much better alternative to synthesizing adders via LUTs.

CARRY Chains These `CARRY4` blocks can be chained across SLICEs to implement wide adders efficiently. The `CARRY4` BELs *must* be chained vertically consecutively across SLICEs as the Carry-In (`CI`) and Carry-Out (`CO`) pins can only be routed this way. A `CARRY4` cell may also directly connect to LUTs and FFs, and should be placed in the same Site whenever possible to minimize wirelength. Shown in figure 13 shows how a `CARRY4` chain and associated LUTs and FFs can be placed across SLICEs.

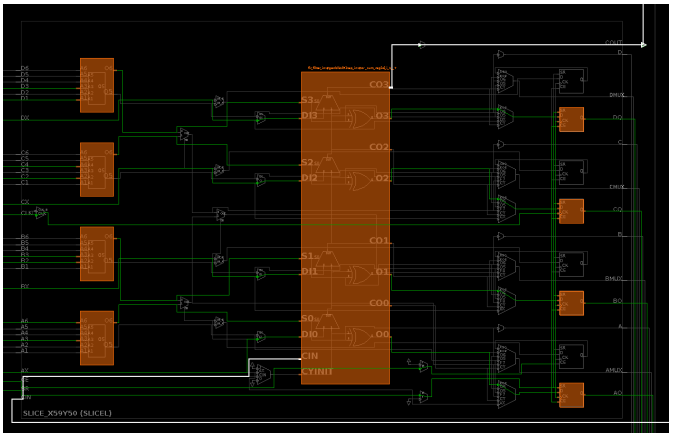


Figure 12: A `SLICEL` with a `CARRY4` cell, 4 LUT cells, and 4 FF cells placed inside as shown in the Vivado device viewer

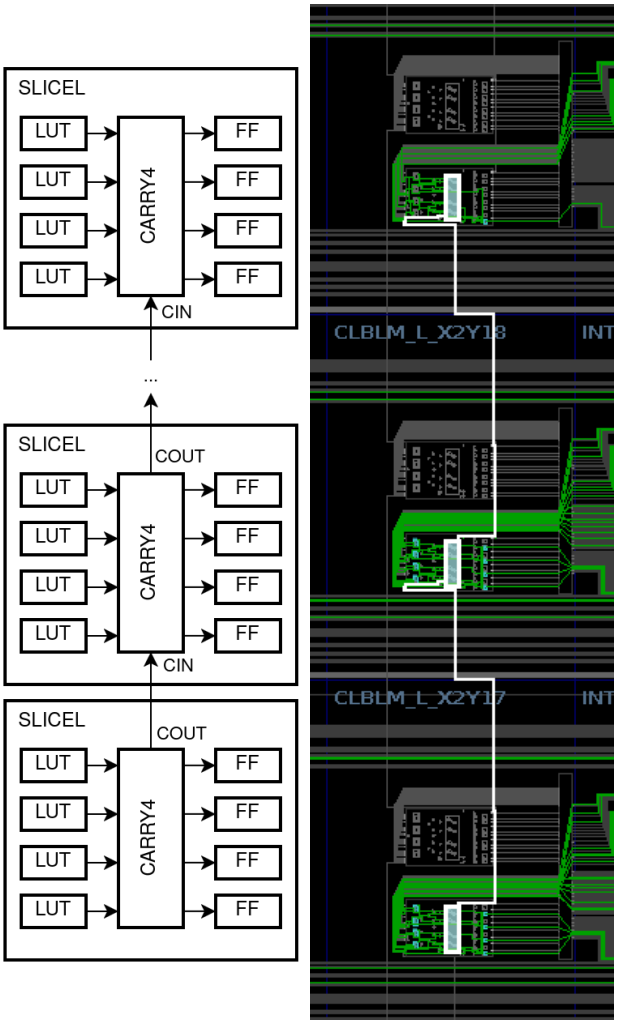


Figure 13: A `CARRY4` chain of size 3 placed across 3 SLICEs. **Left:** Simplified view, **Right:** As shown in the Vivado device viewer.

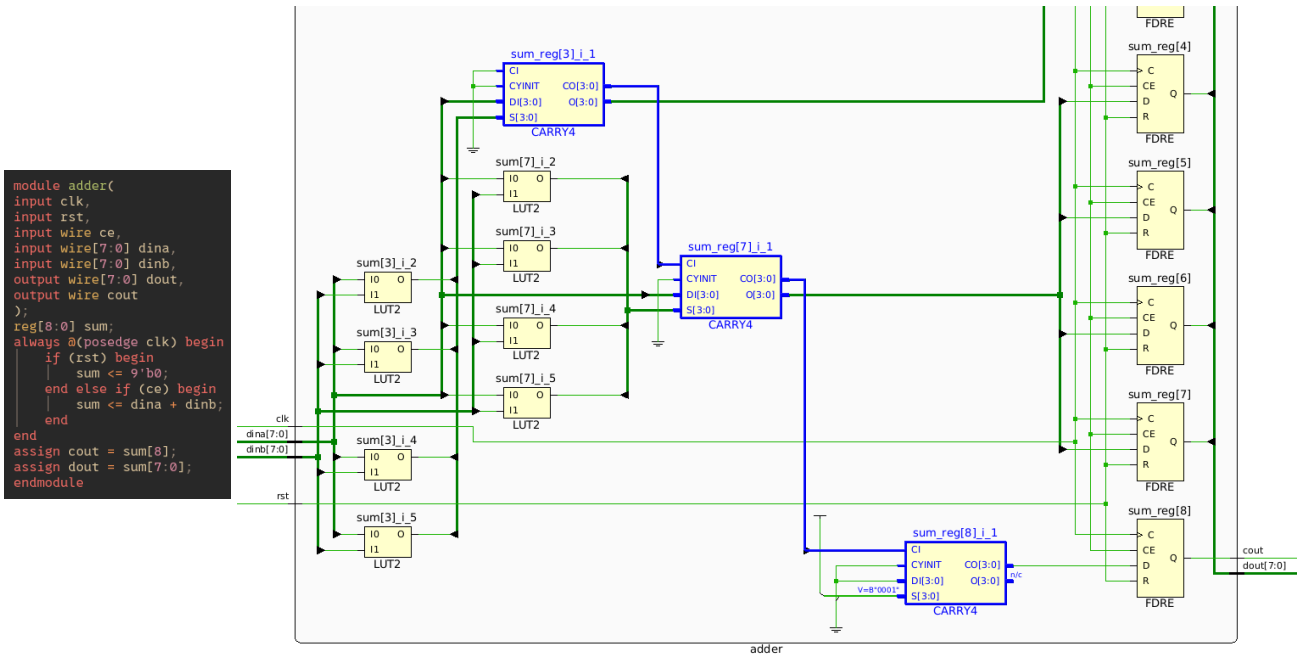


Figure 14: A `CARRY4` chain of size 3 as shown in the Vivado netlist viewer

.

5.2 DSP Slices

DSPs FPGAs are often used as low latency Digital Signal Processing (DSP) accelerators. Common DSP subsystems like Finite Impulse Response (FIR) filters, Fast Fourier Transform (FFTs), and convolutional neural nets (CNNs) demand fast large scale multiply-accumulate (MAC) capabilities. The 7-Series arhictecture integrates DSP BELs into the logic fabric called DSP48E1 that can facilitate MAC efficiently. The architecture hierarchy for DSPs is simple compared to CLBs and SLICEs. A `DSP48E1` Tile contains two `DSP48E1` Sites, each containing one `DSP48E1` BEL, which can host a `DSP48E1` Cell.

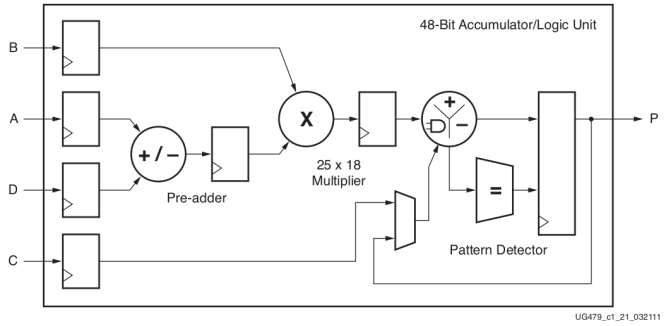


Figure 15: Basic `DSP48E1` Slice Functionality

DSP Cascades Wider DSP functions are supported by cascading `DSP48E1` slices in a `DSP48E1` column. Much like `CARRY4` chains, `DSP48E1` cascades must necessarily be placed vertically consecutively across `DSP48E1` Sites. They are connected by three busses: `ACOUT` to `ACIN`, `BCOUT` to `BCIN`, `PCOUT` to `PCIN`. These signal busses run directly between the vertical `DSP48E1` slices without burdening the general routing resources. The ability to cascade this way provides a high-performance and low-power implementation of digital signal processing systems.

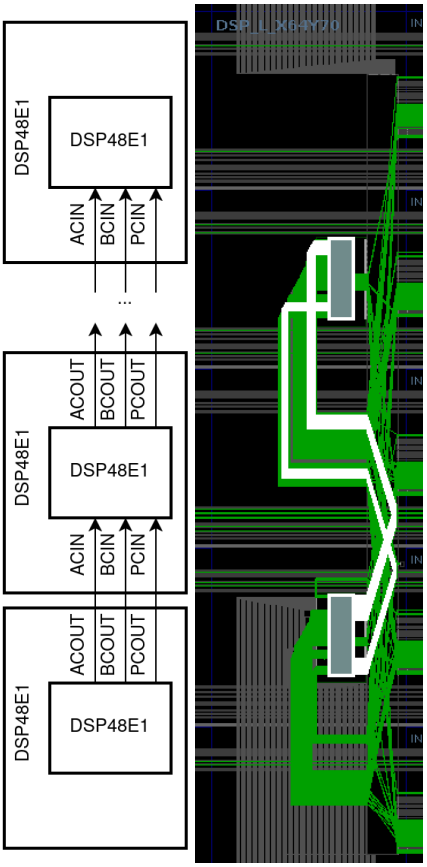


Figure 16: A `DSP48E1` cascade of size 2 placed across 2 `DSP48E1` Sites. **Left:** Simplified view, **Right:** As shown in the Vivado device viewer.

```
module mult(
input wire[23:0] dina,
input wire[23:0] dinb,
output wire[47:0] dout
);
assign dout = dina * dinb;
endmodule
```

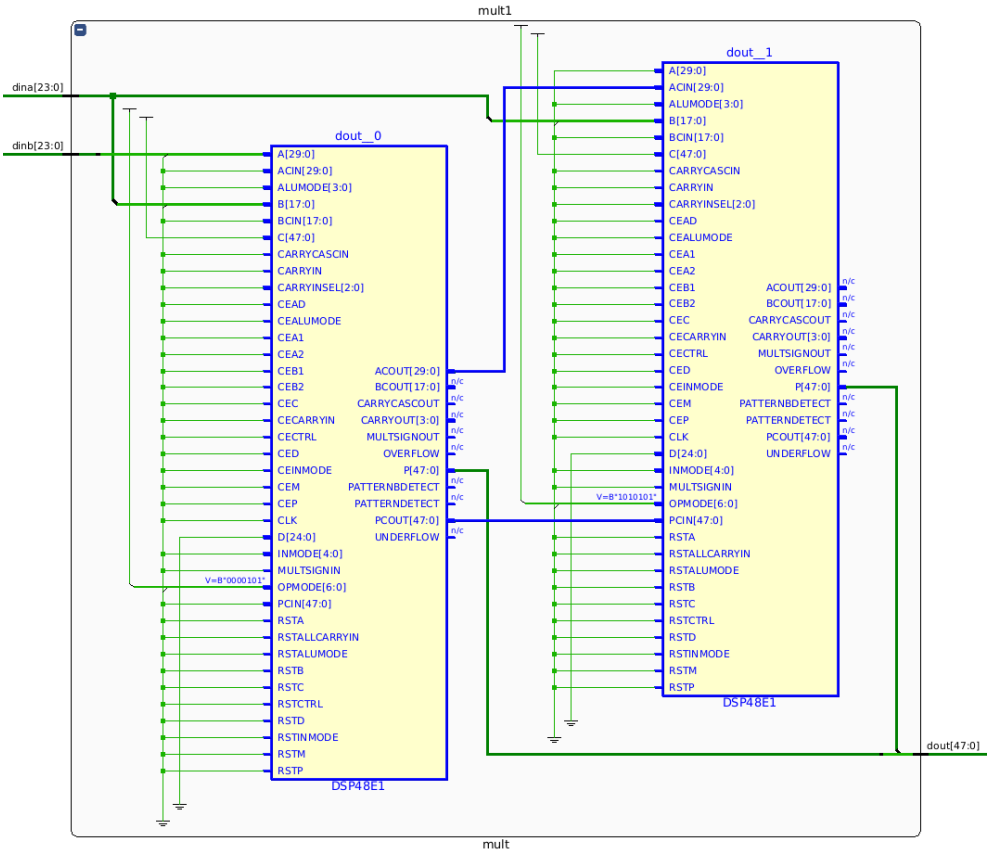


Figure 17: Simple multiplier synthesis from user design.

5.3 Block RAM

In addition to SLICEs and DSPs, the 7-Series also offers dedicated Block Random Access Memory (BRAM) BELs. These BRAMs come in two variants: **RAMB18E1** and **RAMB36E1**.

- `RAMB18E1` - Has a capacity of 18 Kilobits. It can be configured as single port RAM with dimensions ranging between (1-bit wide by 16K deep) to (18-bit wide by 1024 deep). It can also be configured as a (36-bit wide by 512 deep) true simple dual port RAM.

- `RAMB36E1` - Has a capacity of 36 Kilobits. It can be configured as single port RAM with dimensions ranging between (1-bit wide by 32K deep) to (36-bit wide by 1024 deep). It can also be configured as a (72-bit wide by 512 deep) simple dual port RAM.

One BRAM Tile contains one `RAMB36E1` Site and two `RAMB18E1` Sites. The `RAMB36E1` Site contains one `RAMB36E1` BEL, which can host one `RAMB36E1` Cell. Likewise, the `RAMB18E1` Site contains one `RAMB18E1` BEL, which can host one `RAMB18E1` Cell.

Like DSP cascades, BRAMs may also be cascaded together in a column to implement large memories efficiently, with some intermediate signals between them routed intra-Tile without burdening the general routing. However, unlike DSPs, large memories decomposed amongst multiple BRAMs can also be routed together through general routing. Furthermore, in most design scenarios, large memories will not utilize the intra-Tile signals.

To simplify the problem space, we will not constrain large BRAMs to be cascaded together in consecutive vertical Tiles. We can essentially treat `RAMB18E1` and `RAMB36E1` Cells as loose, minimally constrained single cells, in contrast to the highly constrained `DSP48E1` cascades and `CARRY4` chains.

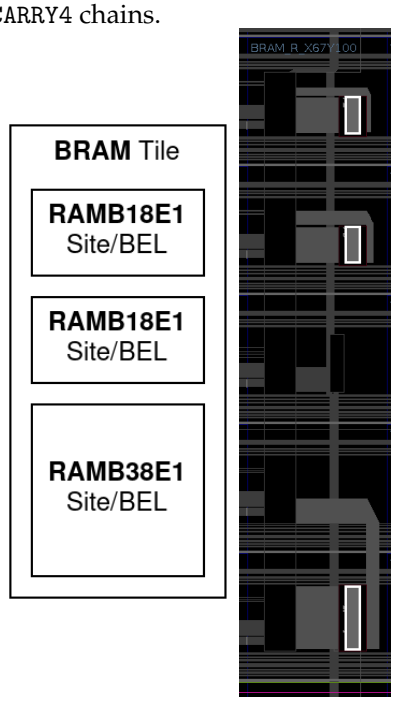


Figure 18: A BRAM Tile containing two `RAMB18E1` Sites and one `RAMB36E1` Site. **Left:** simplified view. **Right:** as seen in the device viewer, BELs highlighted in white.

```
module ram(
    input wire clk,
    input wire en,
    input wire we,
    input wire rst,
    input wire [9:0] addr,
    input wire [23:0] din,
    output wire [23:0] dout
);
reg [23:0] ram [1023:0];
reg [23:0] dout;
always @(posedge clk)
    begin
    if (en) begin
        if (we)
            ram[addr] <= din;
        if (rst)
            dout <= 0;
        else
            dout <= ram[addr];
    end
end
endmodule
```

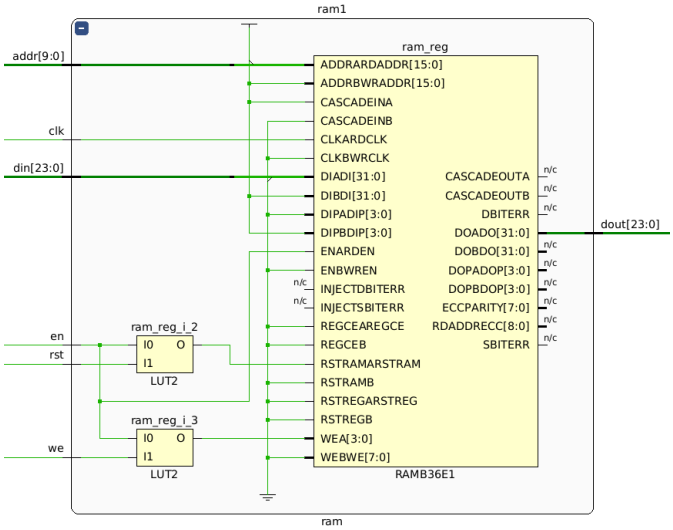


Figure 19: An example of BRAM synthesis (via inference)

5.4 Further Documentation

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

- *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 20 illustrates a representative process that converts an abstract Hardware Description Language (HDL) design into a verified configuration file for a target device.

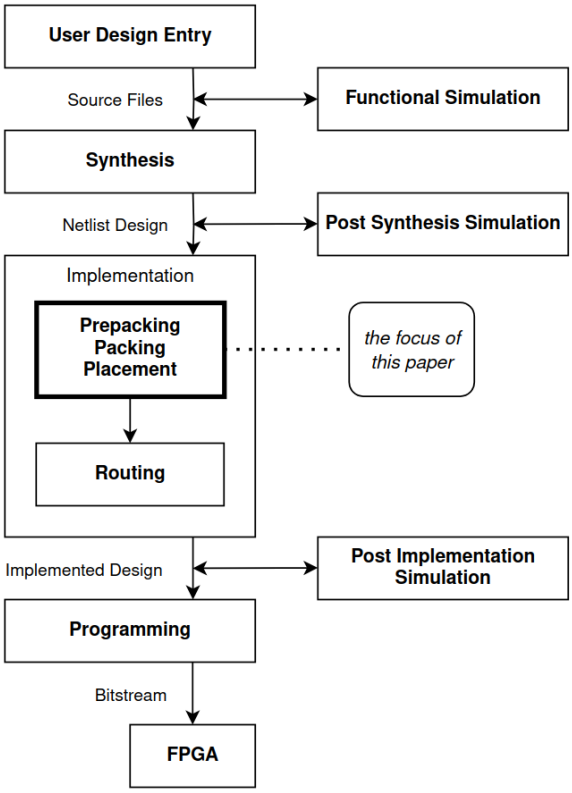


Figure 20: A typical FPGA design and verification workflow.

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 behavioral simulations, then progress to post-synthesis simulations, and finally to post-implementation simulations that incorporate estimated 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.

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 21).

- **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:
  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.

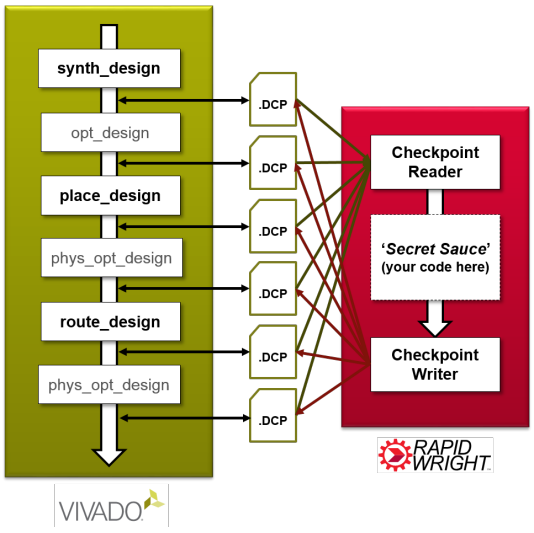


Figure 21: RapidWright workflow integrating into the default Vivado design flow.

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.

7.1 What is a Netlist?

In its most general form, a netlist is a list of every component in an electronic design paired with a list of nets they connect to. Depending on the abstraction level at hand, these components can be transistors, logic gates, macrocells, or increasingly higher-level modules. Generally, a net denotes any group of two or more interconnected components. In an electronics context, a net can be though of as a wire connecting multiple pins between multiple components, with each wire having one voltage source and one or more voltage sinks. Thus, one could express the netlist as a hypergraph, nodes representing components, hyperedges representing wires connecting two or more component. More precisely, these hyperedges connect the ports between the components, not the components themselves, with each component exposing multiple ports.

In FPGA context, the components are logical cells (`LUTs`, `CARRY4s`, etc.) or hierarchical cells (Verilog module instances) with pins connected together by wires. In Vivado, a Netlist can be synthesized as a Hierarchical or a Flattened netlist. Figure 22 shows an example a Verilog design with modules instantiated in a hierarchy. Figure 23 shows the design synthesized into a hierarchical netlist with **hierarchical cells** and **leaf cells**. The synthesizer attempts to construct the module hierarchy as close to the module instantiation hierarchy defined by the user design entry. Figure 24 shows the same design but synthesized into a flattened netlist.

In either synthesized netlist, the **leaf cells**, (deepest level cells), must necessarily consist only of **primitive cells** from the architecture's primitive cell library (`LUT6`, `FDRE`, `CARRY4`, `DSP48E1`, etc.). The netlist can be compiled and exported as a purely structural low-level Verilog file, or an Electrinic Design Interchange Format (EDIF) file, both describing the netlist explicitly as a list of logical cells connected by a list of wires.

```
module top_level(
input wire clk, // clock
input wire rst, // reset
input wire ce,  // clock enable
input wire dina, // data in a
input wire dinb, // data in b
input wire dinc, // data in c
output wire[2:0] dout // data out
);
module_0 m0(
    clk, rst, ce,
    dina, dinb, dinc, dout[0]
);
module_1 m1(
    clk, rst, ce,
    dina, dinb, dinc, dout[2:1]
);
endmodule
```

```
module module_0(
input wire clk, // clock
input wire rst, // reset
input wire ce,  // clock enable
input wire dina, // data in a
input wire dinb, // data in b
input wire dinc, // data in c
output reg dout // data out
);
wire q_0;
reg q_1;
assign q_0 = (dina && !dinb) || dinc;
always @(posedge clk) begin
    if (rst) begin
        dout <= 1'b0;
    end else if (ce) begin
        q_1 <= q_0;
        dout <= q_1;
    end
end
endmodule
```

```
module module_1(
input wire clk, // clock
input wire rst, // reset
input wire ce,  // clock enable
input wire dina, // data in a
input wire dinb, // data in b
input wire dinc, // data in c
output wire[1:0] dout // data out
);
module_2 m2(
    clk, rst, ce,
    dina, dinb, dinc, dout[0]
);
module_3 m3(
    clk, rst, ce,
    dina, dinb, dinc, dout[1]
);
endmodule
```

```
module module_2(
input wire clk, // clock
input wire rst, // reset
input wire ce,  // clock enable
input wire dina, // data in a
input wire dinb, // data in b
input wire dinc, // data in c
output reg dout // data out
);
wire q;
assign q = (!dina && dinb) || dinc;
always @(posedge clk) begin
    if (rst) begin
        dout <= 1'b0;
    end else if (ce) begin
        dout <= q;
    end
end
endmodule
```

```
module module_3(
input wire clk, // clock
input wire rst, // reset
input wire ce,  // clock enable
input wire dina, // data in a
input wire dinb, // data in b
input wire dinc, // data in c
output reg dout // data out
);
wire q;
assign q = (dina && dinb) || !dinc;
always @(posedge clk) begin
    if (rst) begin
        dout <= 1'b0;
    end else if (ce) begin
        dout <= q;
    end
end
endmodule
```

Figure 22: A simple HDL design with module hierarchy.

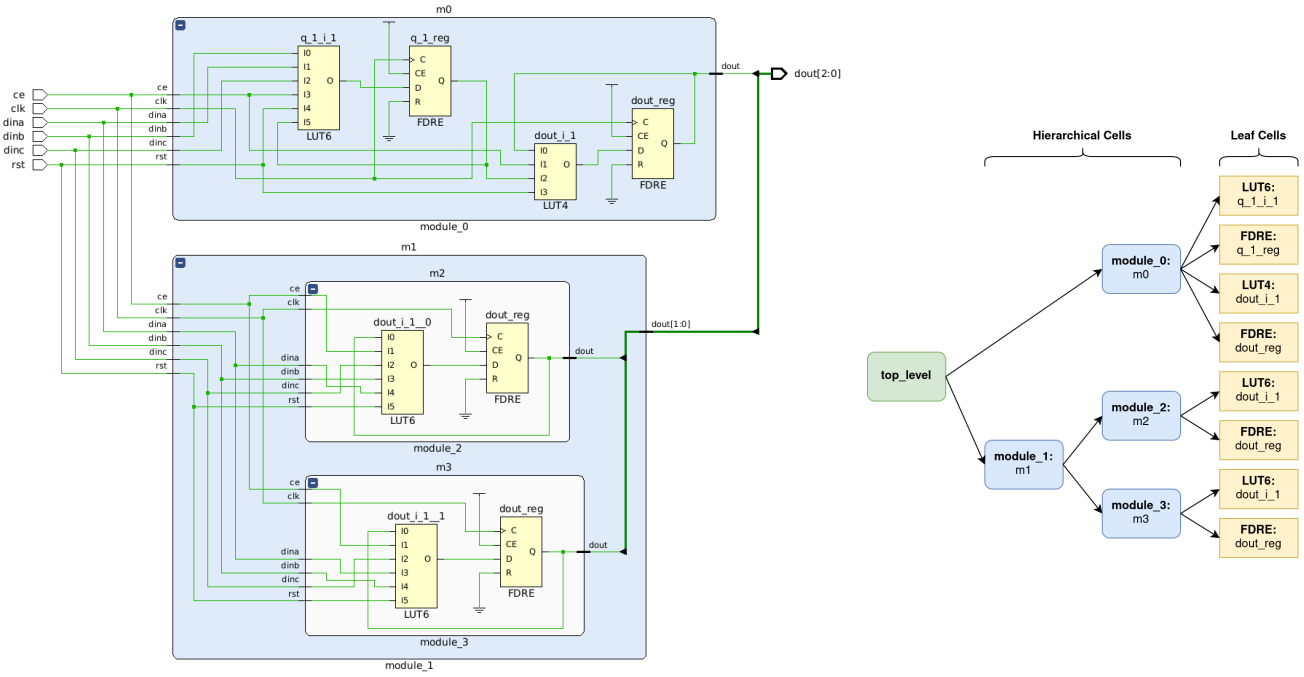


Figure 23: **Left:** A hierarchical netlist consisting of LUTs and FFs. **Right:** The cell hierarchy tree.

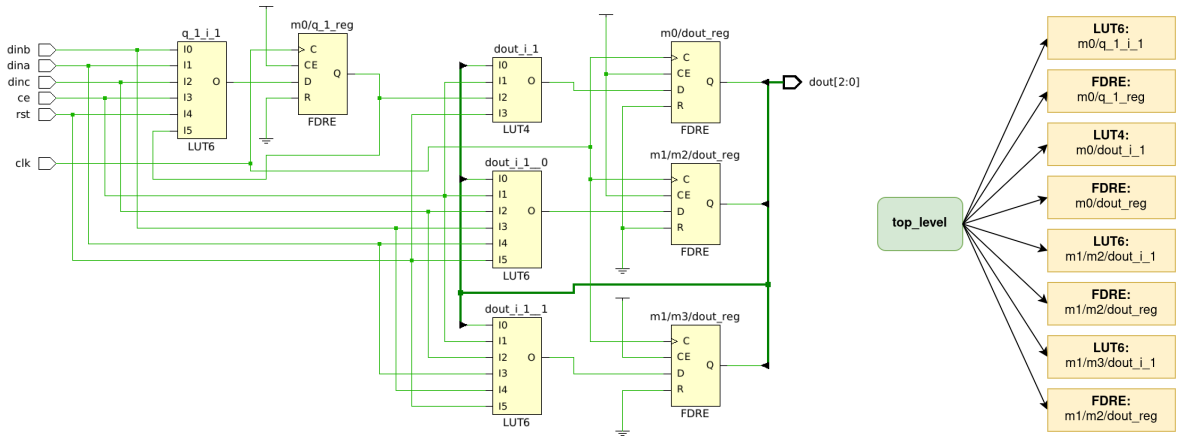


Figure 24: **Left:** A flattened netlist consisting of LUTs and FFs. **Right:** The flattened cell hierarchy tree.

7.2 Netlist Traversal and Manipulation in RapidWright

RapidWright represents the logical netlist objects via the `edif` classes:

- `EDIFNetlist`: The full logical netlist of a `Design`.
- `EDIFNet`: Represents a logical net within an `EDIFNetlist`.
- `EDIFHierNet`: Combines an `EDIFNet` with a full hierarchical instance name to uniquely identify a net in a netlist.
- `EDIFCell`: Represents a logical cell in an `EDIFNetlist`.
- `EDIFCellInst`: Represents an instance of an `EDIFCell`.
- `EDIFHierCellInst`: An `EDIFCellInst` with its hierarchy, described by all the `EDIFCellInsts` that sit above it within the netlist.
- `EDIFPort`: Represents a port on an `EDIFCell`.
- `EDIFPortInst`: Represents an instance of a port on an `EDIFCellInst`.
- `EDIFHierPortInst`: Combines an `EDIFHierPortInst` with a full hierarchical instance name to uniquely identify a port instance in a netlist.

One of RapidWright's most powerful features is netlist traversal and manipulation via the `edif` classes. A netlist can be easily extracted from a `.dcp` design checkpoint and traversed like the following:

Listing 1: Netlist extraction and traversal

```java
Design design = Design.readCheckpoint("synth.dcp")
EDIFNetlist netlist = design.getNetlist();

// Example task:
// Extract the set of all unique nets from the design.

// Initialize a new Set:
Set<EDIFNet> netSet = new HashSet<>();

// Access all leaf cells
List<EDIFCellInst> ecis = netlist.getAllLeafCellInstances();

// Traverse the cell list
for (EDIFCellInst eci : ecis) {
    // Access the ports on this cell
    Collection<EDIFPortInst> epis = eci.getPortInsts();
    for (EDIFPortInst epi : epis) {
        // Access the net on this port
        EDIFNet net = epi.getNet();
        netSet.add(net);
    }
}

// Downstream task:
// For each unique net, print out the connected cells.

// Traverse the set of nets
for (EDIFNet net : netSet) {
    System.out.println("Net: " + net.getName());
    // Access the ports connected to this net
    Collection<EDIFPortInst> epis = net.getPortInsts();
    for (EDIFPortInst epi : epis) {
        // Access the cell that this port belongs to
        EDIFCellInst eci = epi.getCellInst();
        if (eci == null) {
            // (top_level ports have no associated cell)
            continue;
        } else {
            System.out.println(
                "\tCell: " + eci.getName() +
                ",\tCellType: " + eci.getCellName()
            );
        }
    }
}
```

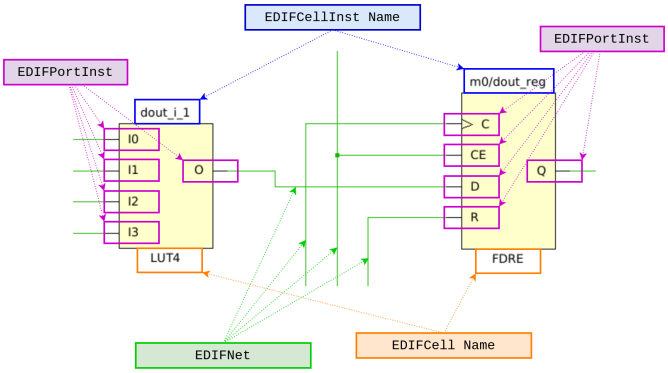


Figure 25: `Netlist` traversal via `EDIFHierCellInst`, `EDIFHierPortInst`, and `EDIFHierNet` objects

Listing 2: Code Printout

```
Net: dout[0]
  Port: I0, Cell: dout_i_1, CellType: LUT4
  Port: Q, Cell: m0/dout_reg, CellType: FDRE
Net: q_1
  Port: I2, Cell: dout_i_1, CellType: LUT4
  Port: Q, Cell: m0/q_1_reg, CellType: FDRE
  Port: I5, Cell: q_1_i_1, CellType: LUT6
Net: ce
  Port: I1, Cell: dout_i_1, CellType: LUT4
  Port: I1, Cell: dout_i_1__0, CellType: LUT6
  Port: I1, Cell: dout_i_1__1, CellType: LUT6
  Port: I3, Cell: q_1_i_1, CellType: LUT6
Net: dout[1]
  Port: I0, Cell: dout_i_1__0, CellType: LUT6
  Port: Q, Cell: m1/m2/dout_reg, CellType: FDRE
Net: dout_i_1__1_n_0
  Port: O, Cell: dout_i_1__1, CellType: LUT6
  Port: D, Cell: m1/m3/dout_reg, CellType: FDRE
Net: clk
  Port: C, Cell: m0/dout_reg, CellType: FDRE
  Port: C, Cell: m0/q_1_reg, CellType: FDRE
  Port: C, Cell: m1/m2/dout_reg, CellType: FDRE
  Port: C, Cell: m1/m3/dout_reg, CellType: FDRE
Net: dout[2]
  Port: I0, Cell: dout_i_1__1, CellType: LUT6
  Port: Q, Cell: m1/m3/dout_reg, CellType: FDRE
Net: <const0>
  Port: G, Cell: GND, CellType: GND
  Port: R, Cell: m0/dout_reg, CellType: FDRE
  Port: R, Cell: m0/q_1_reg, CellType: FDRE
  Port: R, Cell: m1/m2/dout_reg, CellType: FDRE
  Port: R, Cell: m1/m3/dout_reg, CellType: FDRE
Net: <const1>
  Port: P, Cell: VCC, CellType: VCC
  Port: CE, Cell: m0/dout_reg, CellType: FDRE
  Port: CE, Cell: m0/q_1_reg, CellType: FDRE
  Port: CE, Cell: m1/m2/dout_reg, CellType: FDRE
  Port: CE, Cell: m1/m3/dout_reg, CellType: FDRE
Net: dout_i_1__0_n_0
  Port: O, Cell: dout_i_1__0, CellType: LUT6
  Port: D, Cell: m1/m2/dout_reg, CellType: FDRE
Net: rst
  Port: I3, Cell: dout_i_1, CellType: LUT4
  Port: I5, Cell: dout_i_1__0, CellType: LUT6
  Port: I5, Cell: dout_i_1__1, CellType: LUT6
  Port: I4, Cell: q_1_i_1, CellType: LUT6
Net: dinc
  Port: I2, Cell: dout_i_1__0, CellType: LUT6
  Port: I2, Cell: dout_i_1__1, CellType: LUT6
  Port: I2, Cell: q_1_i_1, CellType: LUT6
Net: dinb
  Port: I3, Cell: dout_i_1__0, CellType: LUT6
  Port: I4, Cell: dout_i_1__1, CellType: LUT6
  Port: I0, Cell: q_1_i_1, CellType: LUT6
Net: dina
  Port: I4, Cell: dout_i_1__0, CellType: LUT6
  Port: I3, Cell: dout_i_1__1, CellType: LUT6
  Port: I1, Cell: q_1_i_1, CellType: LUT6
Net: q_1_i_1_n_0
  Port: D, Cell: m0/q_1_reg, CellType: FDRE
  Port: O, Cell: q_1_i_1, CellType: LUT6
Net: dout_i_1_n_0
  Port: O, Cell: dout_i_1, CellType: LUT4
  Port: D, Cell: m0/dout_reg, CellType: FDRE
```

8 Placement

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 26 is an overview of the placement workflow. Figure 27 shows the data structures of RapidWright objects that are populated at each stage.

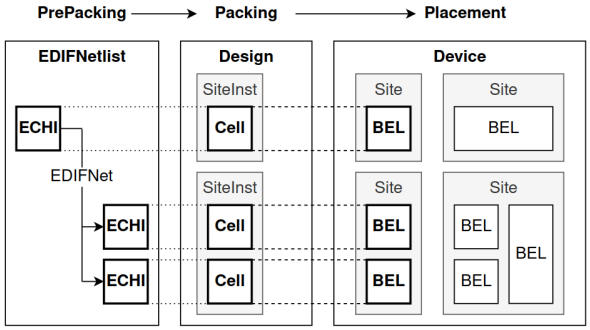


Figure 26: Our placement workflow

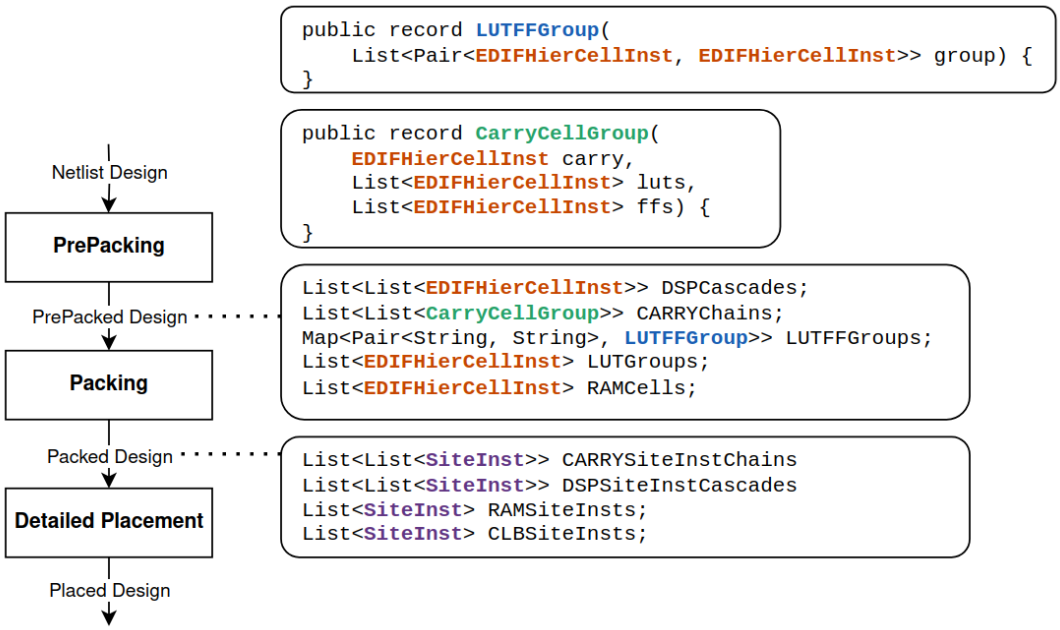


Figure 27: The data classes populated at each substage: PrepackedDesign, PackedDesign, and PlacedDesign.

8.1 Prepacking

The first step in our placement flow is **prepacking**. Recall from the 7-Series architecture that there are certain cell structures that must adhere to certain placements constraints to ensure legality, and by design, to minimize wirelength. The job of the prepacker is to traverse the raw EDIF netlist, detect these cell structures, and consolidate these cells into clusters or groups of clusters that naturally reflect these placement constraints.

Recall that CARRY4 chains must necessarily be placed vertically and consecutively across a column of SLICEs in ascending order. Likewise, `DSP48E1` cascades must necessarily be placed vertically and consecutively across a column of DSP48E1 Sites in ascending order. A LUT-FF pair may be placed freely, but should be placed in the same lane within the same SLICE to minimize wirelength.

The raw EDIF netlist only tells us the list of nets and the cell ports that they connect to. It does not report the presence of any macro cell structures (CARRY4 chains, etc.). Thus, we must traverse the netlist to detect these cell structures and store that structure information in a class we will call `PrepackedDesign`.

A group of LUT-FF pairs may be placed in the same SLICE, with the constraint that the FFs must share the same Clock-Enable (CE) and Set-Reset (SR) nets. Clustering LUT-FF pairs like this can reduce the redundancy of having to route the same CE and SR nets to many SLICEs by routing the nets to fewer SLICEs and connecting them to the individual FFs within via intra-Site routing, and at the same time, packing greater logic density over a smaller area of the device. Recall that up to eight FFs may be placed in the same SLICE, thus theoretically, up to eight LUT-FF pairs can be placed in the same SLICE.

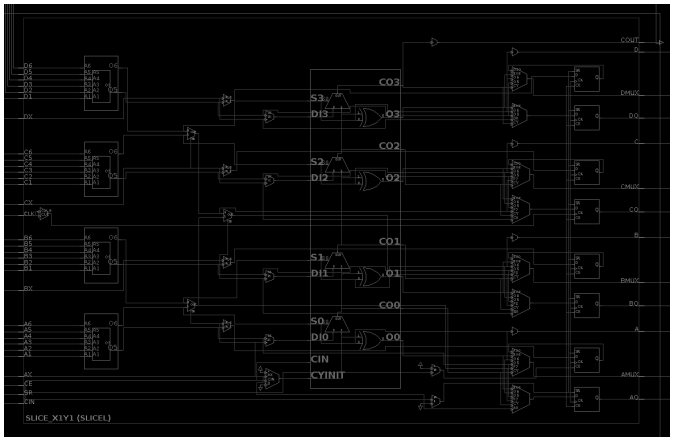


Figure 28: A SLICEL Site

Utilizing all 8 LUT-FF lanes in a SLICE can help reduce device area utilization and minimize wirelength, but *too much* logic density in an area can contribute to general routing congestion. Furthermore, attempting to fill all 8 lanes in a SLICE requires meticulous adherence

to conditional LUT constraints. Recall that a LUT can accommodate one 6-input boolean function or two 5-input boolean functions sharing the same inputs or any two 3-input or less boolean functions regardless of shared inputs. A LUT6 Cell (a LUT with 6-inputs) will actually occupy two LUT-FF lanes in a SLICEL, rendering one of the FF BELs in either lane ineligible for another LUT-FF pair. To simplify the problem space, we will only fill up to four LUT-FF lanes in any given SLICE.

Listing 3: Netlist extraction and traversal

```
Design design = Design.readCheckpoint("synth.dcp")
EDIFNetlist netlist = design.getNetlist();
List<EDIFCellInst> ecis = netlist.getAllLeafCellInstances();

// Select only the carry cells.
for (EDIFCellInst eci : ecis) {
    if (eci.getCellName() == "CARRY4")
        carryCells.add(eci);
}

// Find and remove carry chains until the list is empty
while (!carryCells.isEmpty()) {
    // Arbitrarily set "currentCell" pointer to a cell in the
     list
    EDIFCellInst currentCell = carryCells.get(0);

    // Find this carry chain anchor.
    // Traverse the Carry-In (CI) to Carry-Out (CO) nets.
    // Anchor is found when net on the CI Port is Ground.
    while (true) {
        // Access the CI port on this cell.
        EDIFPortInst sinkPort = currentCell.getPortInst("CI");
        // Access the net on this CI port.
        EDIFNet net = sinkPort.getNet();
        if (net.isGND())
            // Found this chain anchor!
            break;
        // Get all ports on this net.
        List<EDIFPortInst> netPorts = net.getPortInsts();
        for (EDIFPortInst netPort : netPorts) {
            // Access the port belonging to another carry cell.
            EDIFCellInst sourceCell = netPort.getCellInst();
            if (sourceCell.getCellName() == "CARRY4") {
                // Move the "currentCell" pointer
                currentCell = sourceCell;
            }
        }
    }

    // Now we have the chain anchor as currentCell.
    // Now traverse in the opposite direction to find the
     chain tail.
    // Tail is found when the CO Port is null.
    // Collect the chain cells into an ordred list.
    List<EDIFCellInst> currentChain = new ArrayList<>();
    while (true) {
        EDIFPortInst sourcePort =
      currentCell.getPortInst("CO[3]");
        if (sourcePort == null)
            // Found this chain's tail!
            break;
        EDIFNet net = sourcePort.getNet();
        List<EDIFPortInst> sinkPorts = net.getPortInsts();
        for (EDIFPortInst sinkPort : sinkPorts) {
            EDIFCellInst sinkCell = sinkPort.getCellInst();
            if (sinkCell.getCellName() == "CARRY4") {
                currentCell = sinkCell;
                // Add the cell to the chain list.
                currentChain.add(currentCell);
            }
        }
    }

    // Add currentChain to the list of chains
    carryChains.add(currentChain);
}
```

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.