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 .

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.

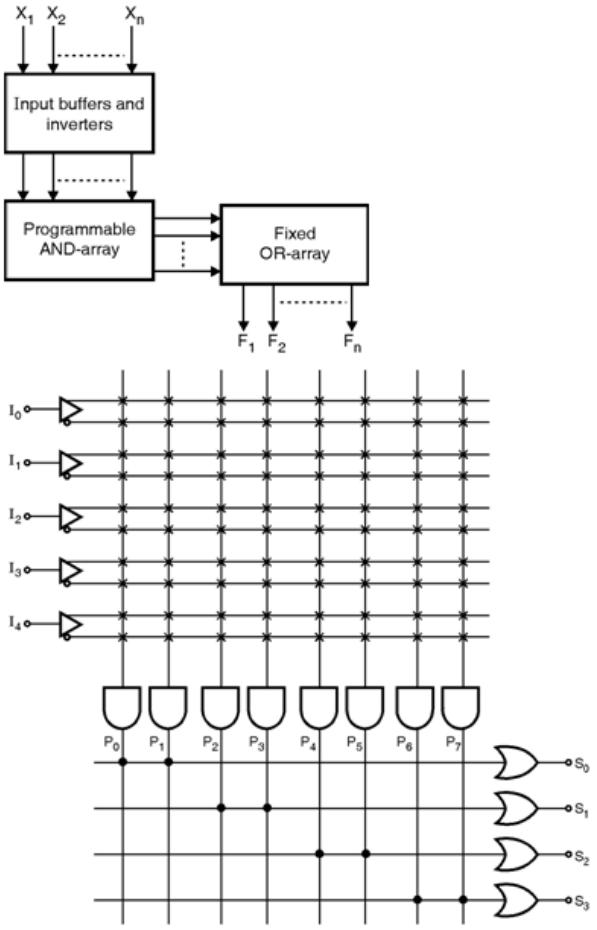


Figure 1: PAL architecture with 5 inputs, 8 programmable AND gates and 4 fixed OR gates

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.

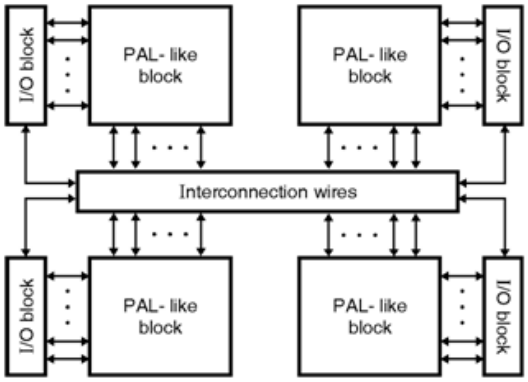


Figure 2: CPLD architecture with 4 CLBs (PAL-like blocks)

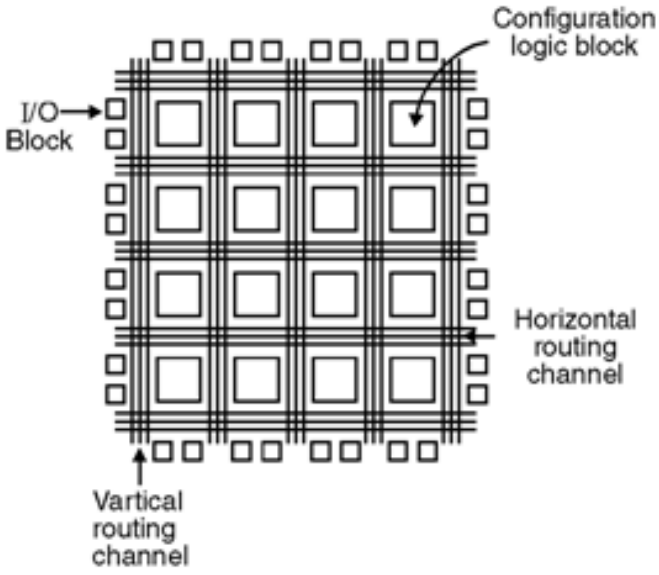


Figure 3: A homogeneous island-style FPGA architecture with 16 CLBs in a grid.

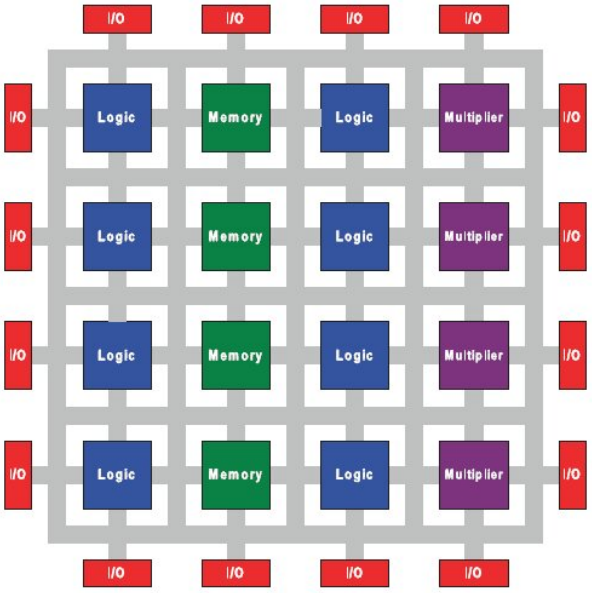


Figure 4: A heterogeneous island-style FPGA with a mix of CLBs and macrocells.

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 (AMD) and Altera (Intel) 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.

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**. Note how the Tile arrangement is columnar such that column consists of only one Tile type. 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.

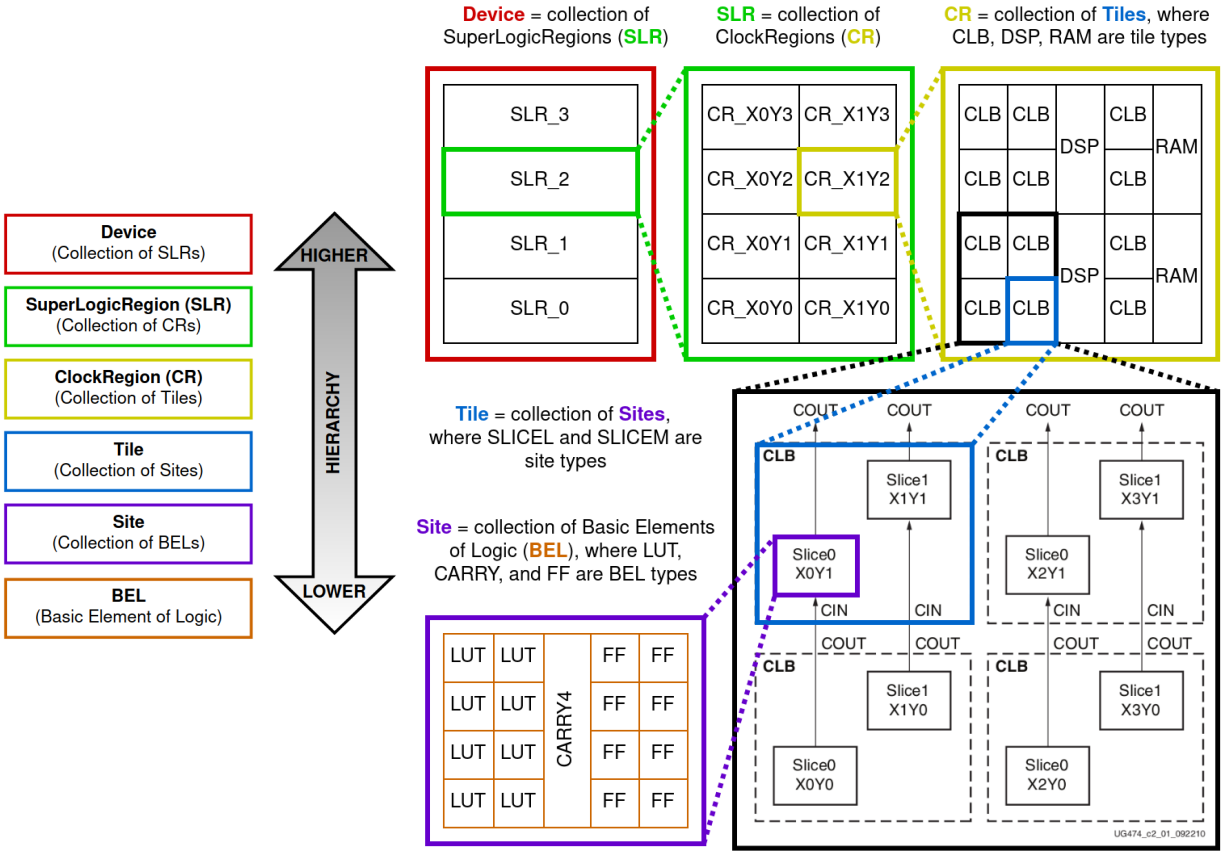


Figure 5: Architecture Hierarchy of a Xilinx FPGA

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 or when the design inheritly demands homogeneously distributed local memory.

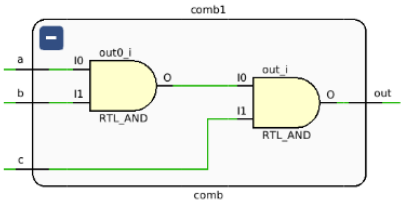
In a typical 7-Series device, approximately 75% of the SLICE Sites are `SLICEL`s and 25% are `SLICEM`s. A single CLB Tile can therefore host either two `SLICEL`s or one `SLICEL` and one `SLICEM`. To simplify the problem space, however, we will only consider `SLICEL`s for general logic and use the dedicated `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 the name *Field Programmable Gate Array* might suggest.

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.

```
module comb(
input wire a,
input wire b,
input wire c,
output wire out
);
assign out = a & b & c;
endmodule
```



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

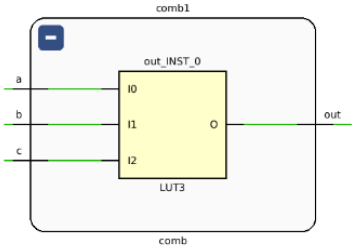


Figure 6: LUT synthesis from user design

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 `FDRE`s 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.

```
module seq(
input wire clk, // clock
input wire rst, // reset
input wire ce,  // clock enable
input wire din, // data in
output reg dout // data out
);
always @(posedge clk) begin
    if (rst) begin
        dout <= 1'b0;
    end else if (ce) begin
        dout <= din;
    end
end
endmodule
```

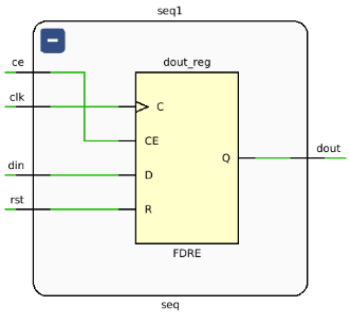


Figure 7: FF synthesis from user design

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.

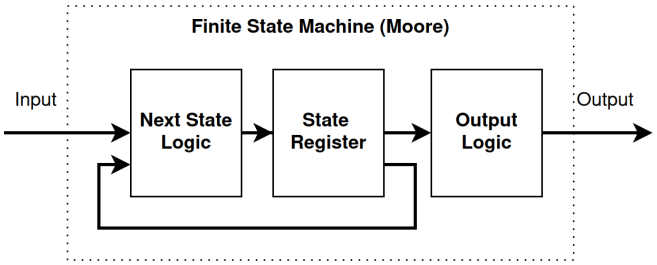


Figure 8: Finite state machine (Moore)

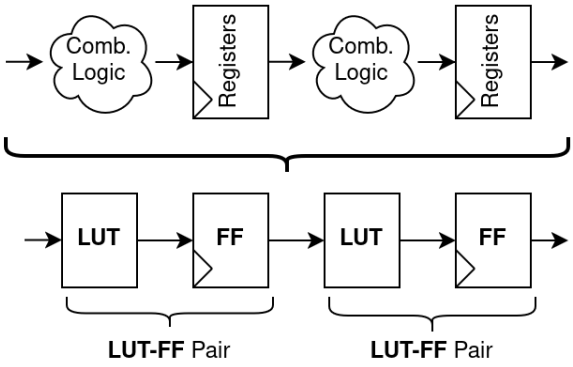


Figure 9: Pipelining synthesized as consecutive LUT-FF pairs

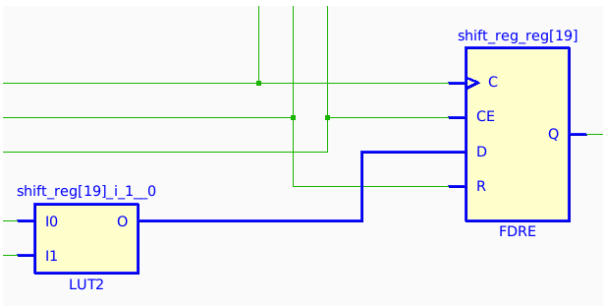


Figure 10: A synthesized LUT-FF Pair

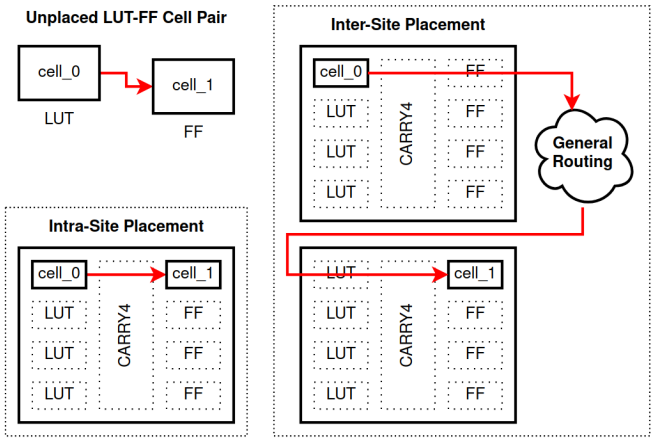


Figure 11: Intrasite vs Intersite LUT-FF Placement

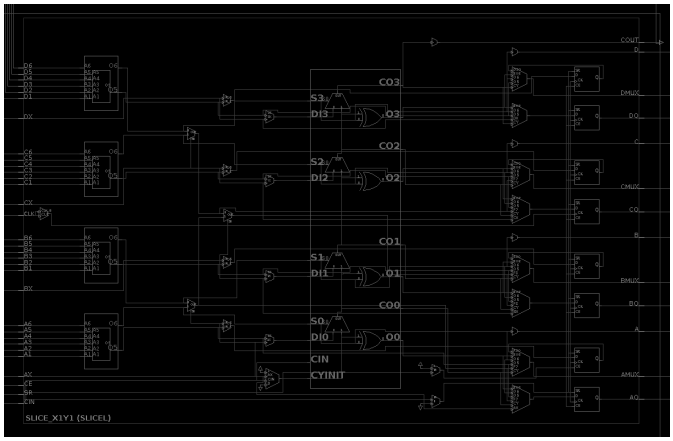


Figure 12: A SLICEL Site

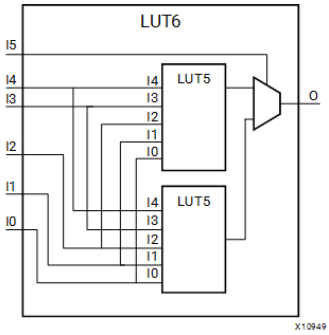


Figure 13: A LUT6 Cell is hosted using two LUT5 BELs.

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.

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 in-

puts. A LUT6 Cell (a LUT with 6-inputs) will actually occupy two LUT-FF lanes in a SLICE, rendering one of the FF BELs in either lane ineligible for another LUT-FF pair. Figure 12 hints at this by depicting pairs of LUTs stacked ontop of one another. The LUTs individually can host up to a LUT5, but combined together can host one LUT6, like shown in Figure 13. To simplify the problem space, we will only fill up to four LUT-FF lanes in any given SLICE.

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 15 shows how a CARRY4 chain and associated LUTs and FFs can be placed across SLICEs.

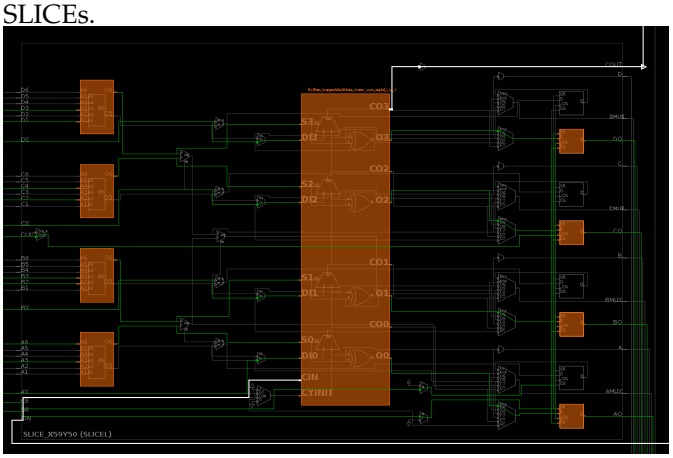


Figure 14: A SLICEL populated by a CARRY4 cell, 4 LUT cells, and 4 FF cells

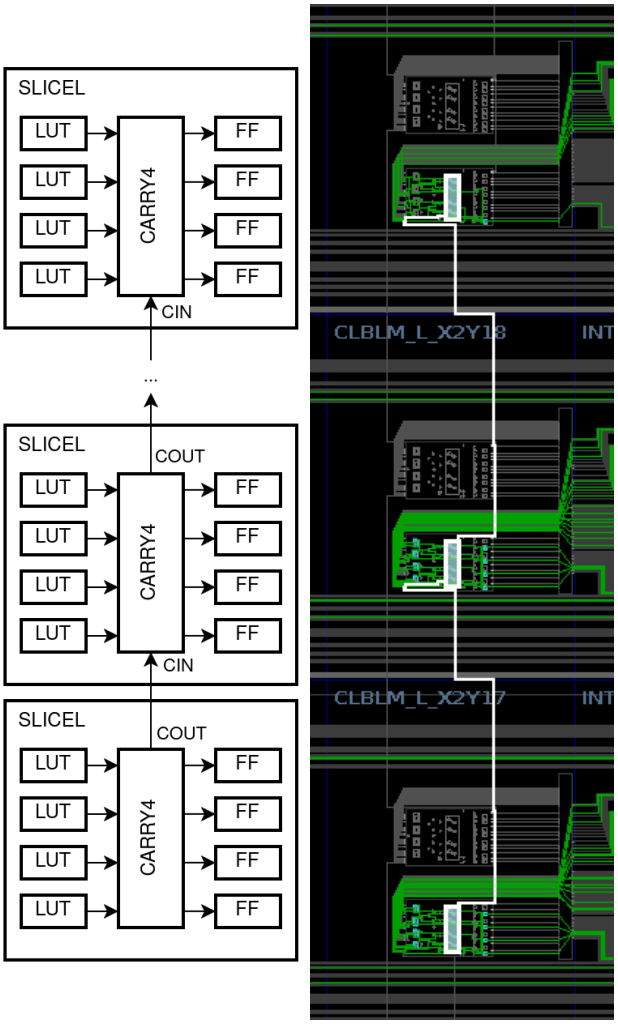


Figure 15: A CARRY4 chain of size 3 placed across 3 SLICEs. **Left:** Simplified view, **Right:** As shown in the Vivado device viewer.

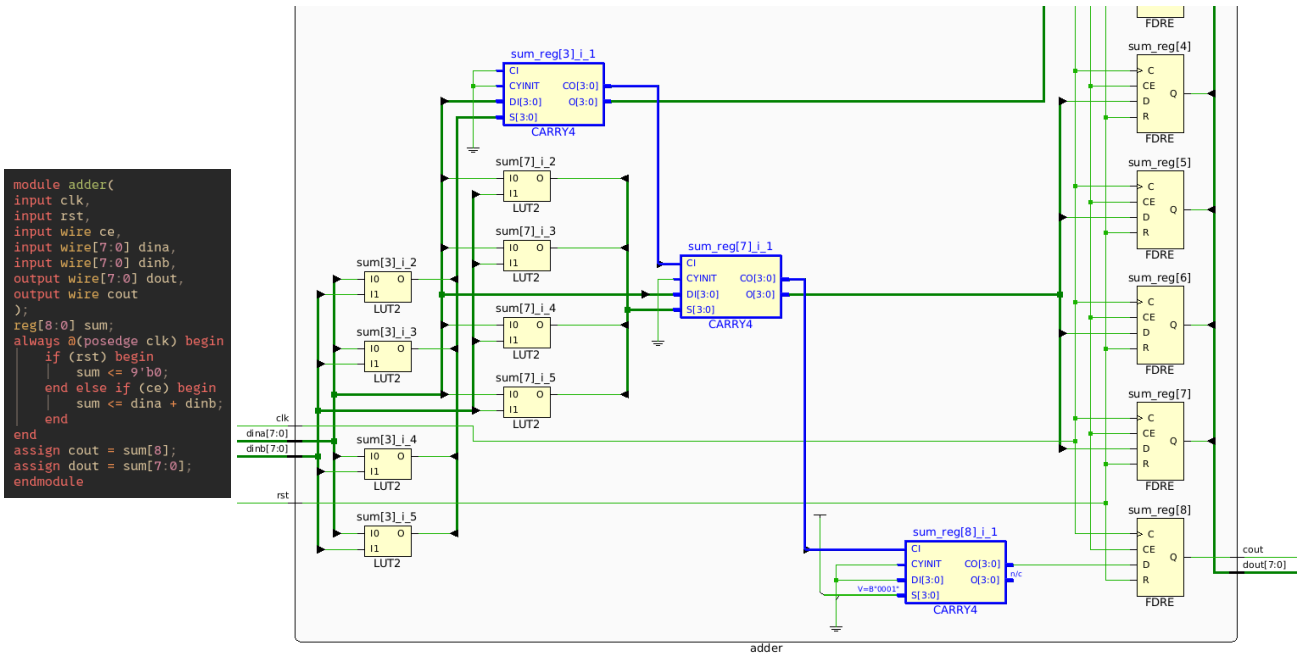


Figure 16: A CARRY4 chain of size 3 as shown in the Vivado netlist viewer

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.

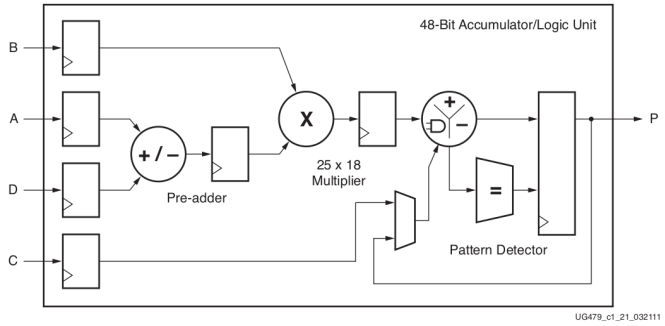


Figure 17: Basic `DSP48E1` Slice Functionality

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.

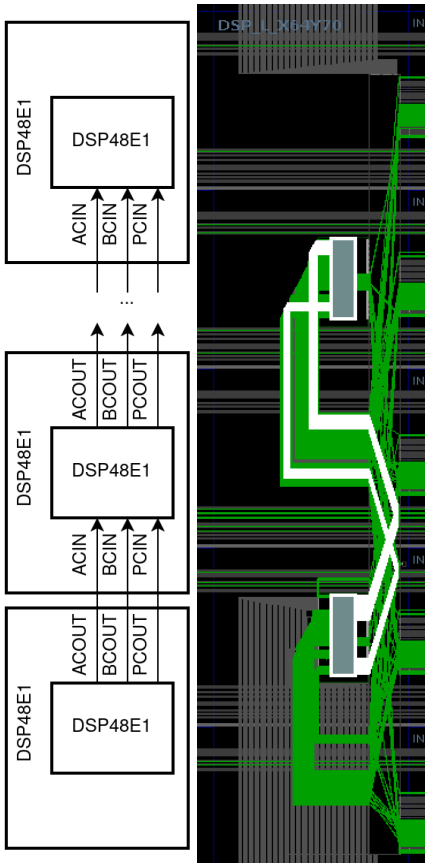


Figure 18: A `DSP48E1` cascade of size 2 placed across 2 `DSP48E1` Sites. **Left:** Simplified view, **Right:** As shown in the Vivado device viewer.

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

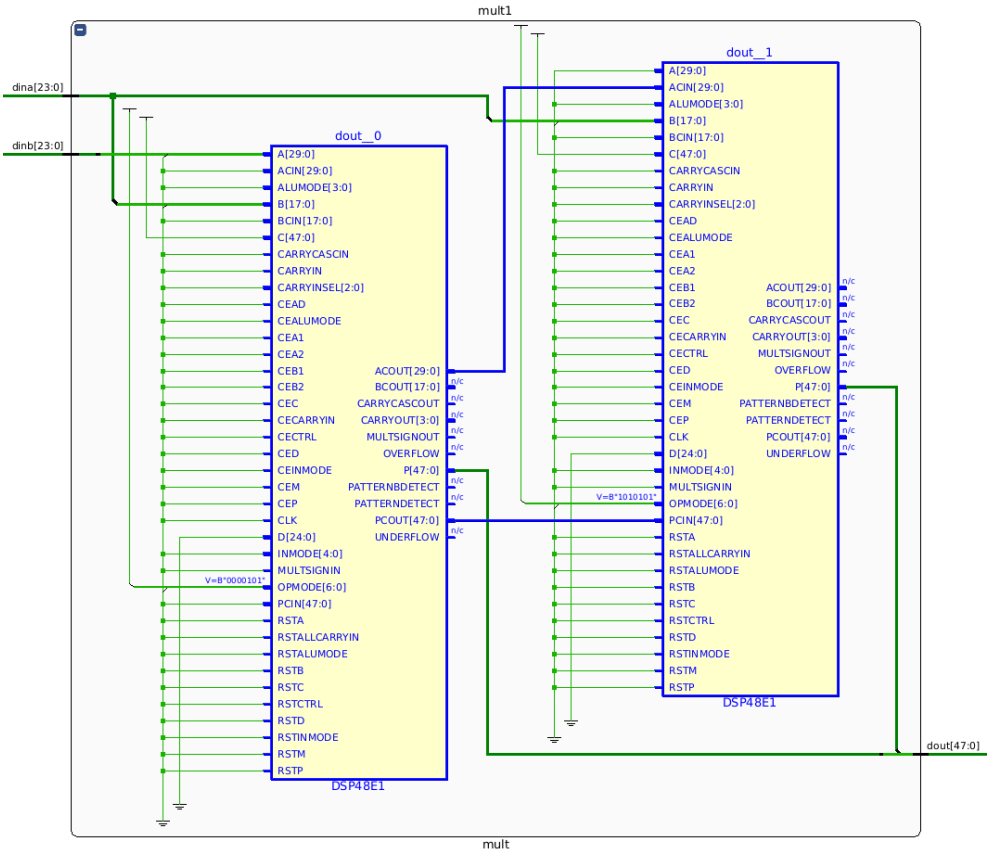


Figure 19: Simple multiplier synthesis from user design.

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.

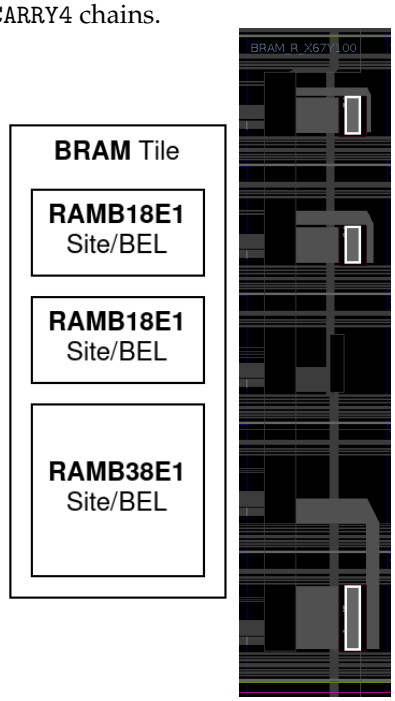


Figure 20: A BRAM Tile containing two `RAMB18E1` Sites and one `RAMB36E1` Site. **Left:** simplified view. **Right:** as seen in the device viewer, BELs highlighted in white.

```
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
```

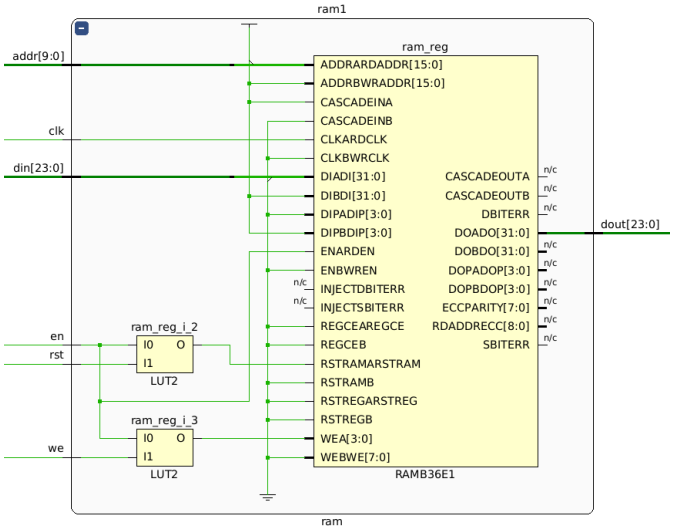


Figure 21: An example of BRAM synthesis (via inference)

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 and placement constraints and optimize wirelength in modern FPGA architectures.

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

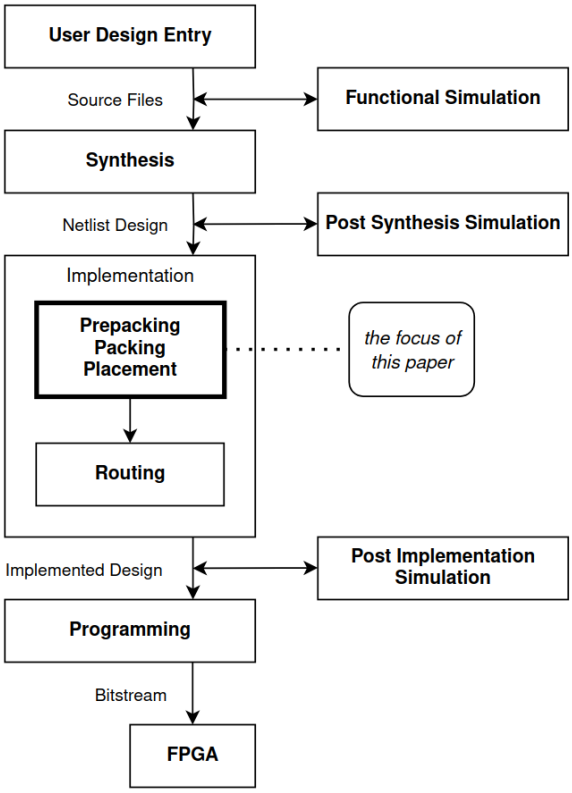


Figure 22: A typical FPGA design and verification workflow.

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 generally 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 23).

- **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.

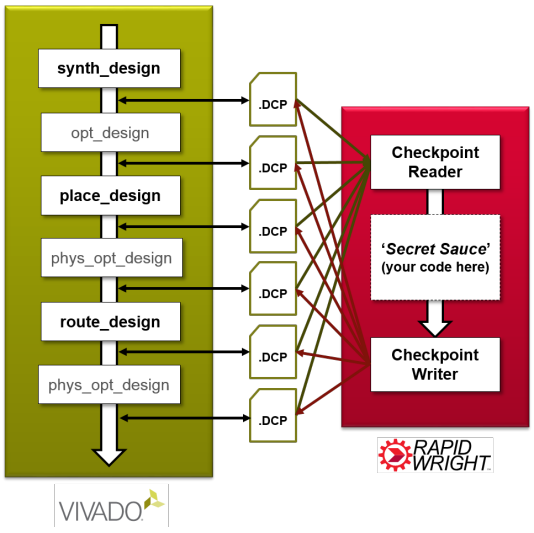


Figure 23: RapidWright workflow integrating into the default Vivado design flow.

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 on 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 24 shows an example a Verilog design with modules instantiated in a hierarchy. Figure 25 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 26 shows the same design but synthesized into a **flattened netlist** containing **only leaf cells**.

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 cell instances 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 24: A simple HDL design with module hierarchy.

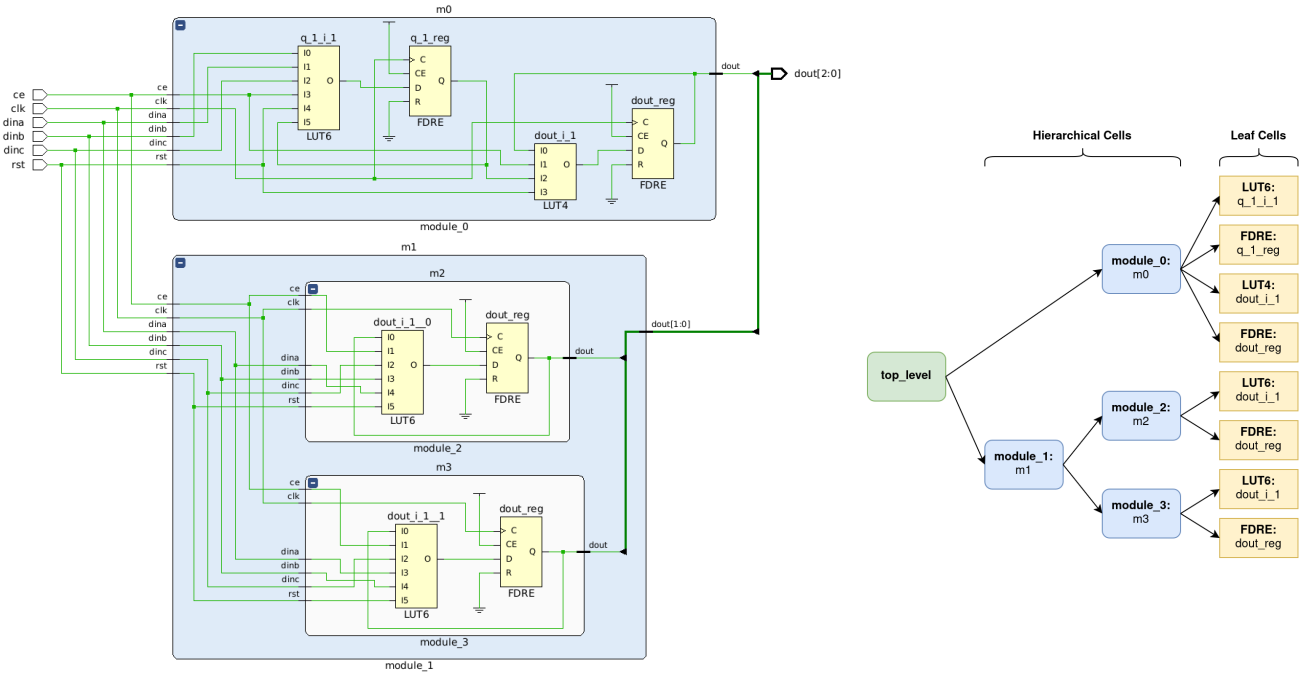


Figure 25: **Left:** A hierarchical netlist consisting of LUTs and FFs. **Right:** The cell hierarchy tree.

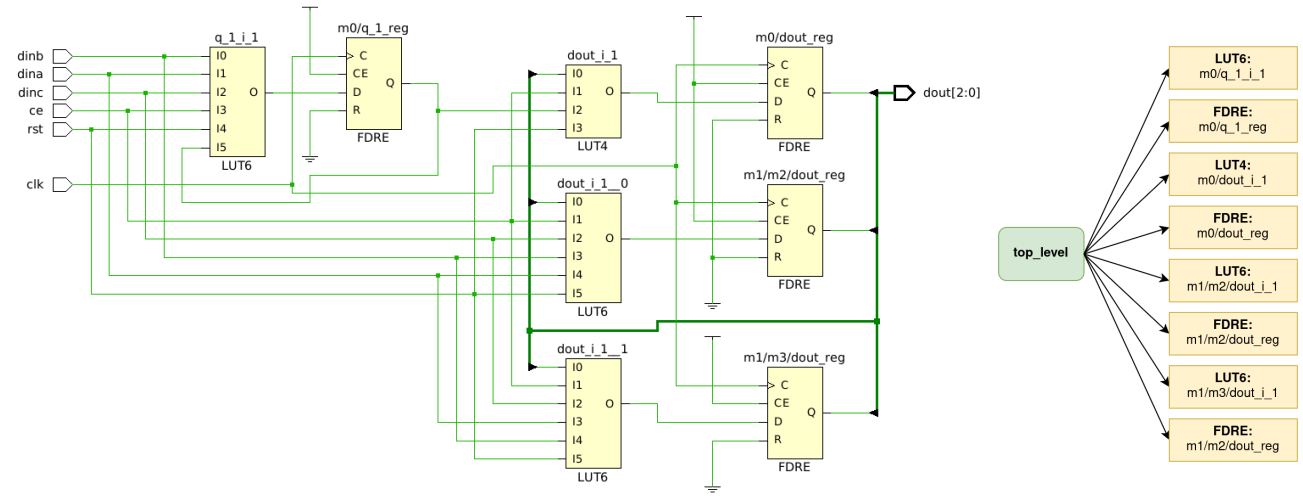


Figure 26: **Left:** A flattened netlist consisting of LUTs and FFs. **Right:** The flattened cell hierarchy tree.

7.2 Netlist Traversal and Manipulation via RapidWright

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

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

Using these classes and their associated methods, we can traverse the logical netlist (`EDIFNetlist`) and analyze or manipulate it as we see fit. A netlist can be easily extracted from a `.dcp` design checkpoint file and traversed like shown in Listing 1. This is performed on the same design shown in figure 26.

Listing 1: Basic netlist extraction and traversal

```
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 incident 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()
            );
        }
    }
}
```

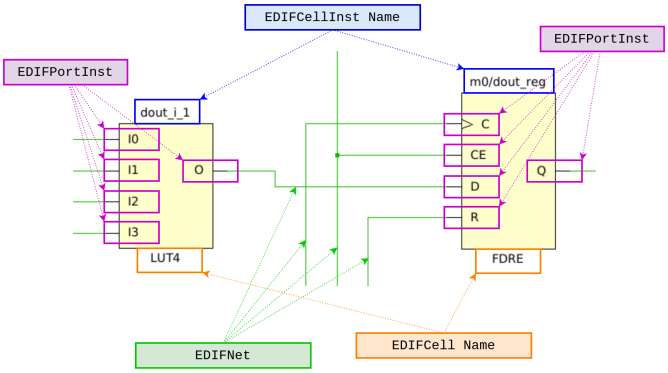


Figure 27: `Netlist` traversal via the `EDIFCellInst`, `EDIFPortInst`, and `EDIFNet` classes

In this method, we get the list of all `EDIFCellInst` from the `EDIFNetlist` and traverse the list of cells and access their connected nets. Alternatively, we could have also extracted the list of all `EDIFNets` directly from the `design` and traversed the nets to access their connected cells.

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
```

7.3 Placement Flow via RapidWright

Figure 28: The data classes populated at each substage: `PrepackedDesign`, `PackedDesign`, and `PlacedDesign`.

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 29 is an overview of the placement workflow. Figure 28 shows the data structures of RapidWright objects that are populated at each stage: `PrepackedDesign`, which is a group of data structures around `EDIFHierCellInsts`, `PackedDesign`, which is a group fo data structures around `SiteInsts`, and finally, `PlacedDesign`, which is simply captured by the final RapidWright `Design` object.

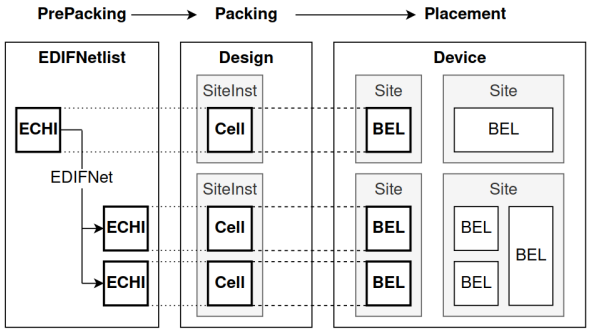


Figure 29: Our placement workflow

8 Prepacking

The first step in our placement flow is **prepacking**. Recall from the 7-Series architecture that there are certain multi-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 multi-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 multi-cell structures (`CARRY4` chains, etc.). Thus, we must traverse the netlist to detect these multi-cell structures and store that structure information in a class we will call `PrepackedDesign`.

The code snippet in 7 shows how one can detect and collect these `CARRY4` chains using RapidWright. We first collect the cells in the design that are of type `CARRY4`, then iteratively traverse their Carry-Out (`CO`) to Carry-In (`CI`) nets to find incident `CARRY4` cells. Each `CARRY4` chain has an anchor cell and a tail cell where the chain terminates. The anchor is found when the `CI` net connects to Ground (`GND`), while the tail is found when the `CO` port is null. We can further detect if there are `LUTs` or `FFs` connected to the `CARRY4` cell and store that information in a data structure we will call `CarryCellGroup` as defined in figure 28. This will help us in knowing which cells can be packed together into the same `Site` in the subsequent stages.

Similarly, `DSP48E1` cascades can be found and collected by traversing the `PCOUT ACOUT` and `BCOUT` nets. `LUT-FF` pairs can be found by inspecting the LUT output (O) net and checking for FF input (DI) ports. We can bucket these LUT-FF pairs by finding the set of unique `CE SR` net pairs to know which group of LUT-FF pairs can be placed within the same `Site`.

The overall goal of prepacking is to detect the presence of these multi-cellular structures and consolidate that information into our `PrepackedDesign` object in preparation for the following Packing stage.

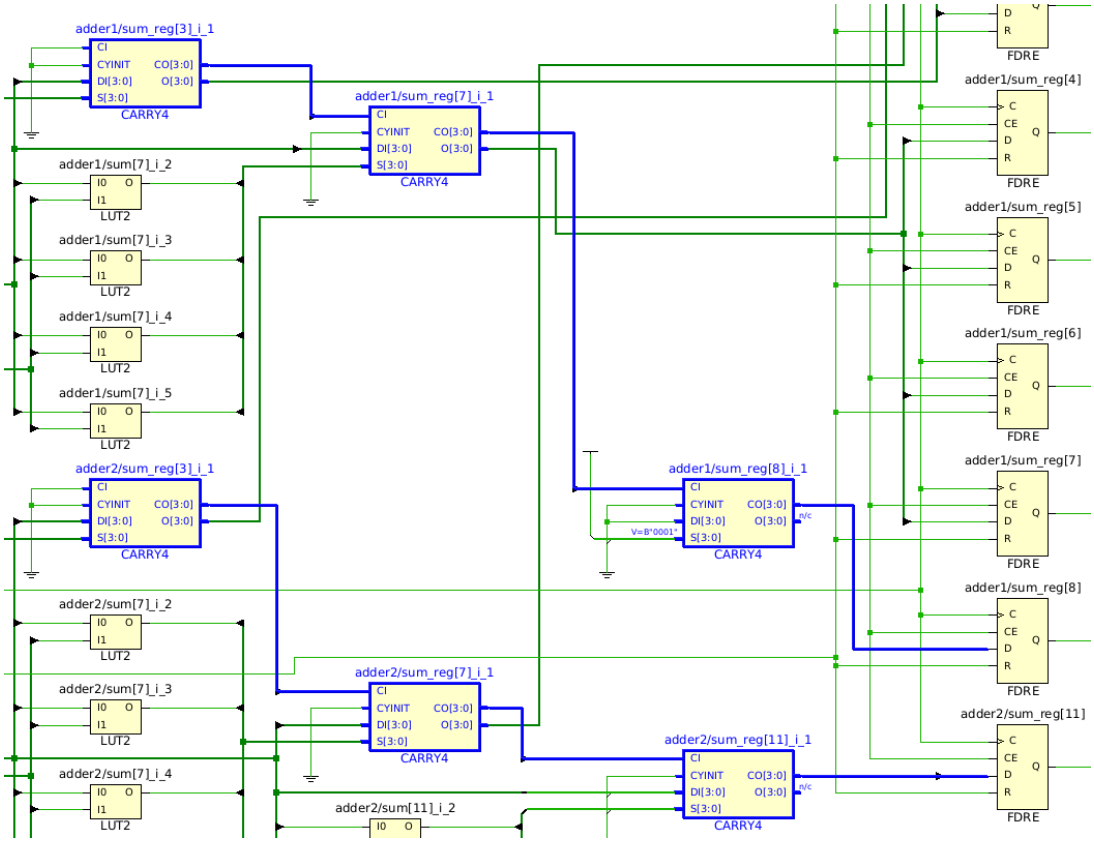


Figure 30: A netlist with two CARRY4 chains, each of size 3

Listing 3: Finding and storing carry chains.

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

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

// Find and remove carry chains until the list is empty
List<List<EDIFCellInst>> carryChains = new ArrayList<>();
while (!carryCells.isEmpty()) {
    // Set "currentCell" pointer to an arbitrary cell in 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) {
        System.out.println(currentCell);
        // 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()) {
            break; // Found this chain anchor!
        }
        // 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().equals("CARRY4")) {
                // Move the "currentCell" pointer
                currentCell = sourceCell;
                break;
            }
        }
    }

    // Now currentCell points at this chain's anchor.

    // 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<>();
    currentChain.add(currentCell);
    while (true) {
        EDIFPortInst sourcePort =
     currentCell.getPortInst("CO[3]");
        if (sourcePort == null) {
            break; // Found this chain's tail!
        }
        EDIFNet net = sourcePort.getNet();
        List<EDIFPortInst> netPorts = net.getPortInsts();
        for (EDIFPortInst netPort : netPorts) {
            EDIFCellInst sinkCell = netPort.getCellInst();
            if (netPort.getName().equals("CI") &&
                    sinkCell.getCellName().equals("CARRY4")) {
                currentCell = sinkCell;
                // Add the cell to the chain list.
                currentChain.add(currentCell);
                break;
            }
        }
    }
    // Add currentChain to the list of chains
    carryChains.add(currentChain);
    // Remove currentChain from the list of cells
    carryCells.removeAll(currentChain);
} // end while()

// Print out the carry chains.
for (List<EDIFCellInst> chain : chains) {
    for (int i = 0; i < chain.size(); i++) {
        EDIFCellInst carry = chain.get(i);
        if (i == 0) {
            writer.write("\nAnchor Cell: " + carry.getName() +
                ", CellType: " + carry.getCellName());
        } else {
            writer.write("\n\tCell: " + carry.getName() +
                ", CellType: " + carry.getCellName());
        }
    }
}
```

Listing 4: Code Printout

```
Anchor Cell: adder2/sum_reg[3]_i_1, CellType: CARRY4
  Cell: adder2/sum_reg[7]_i_1, CellType: CARRY4
  Cell: adder2/sum_reg[11]_i_1, CellType: CARRY4
Anchor Cell: adder1/sum_reg[3]_i_1, CellType: CARRY4
  Cell: adder1/sum_reg[7]_i_1, CellType: CARRY4
  Cell: adder1/sum_reg[8]_i_1, CellType: CARRY4
```

9 Packing

Now that we have our PrepackedDesign object keeping track of multi-cell structures on the edif level, we can start packing them into SiteInst objects on the design level. Below are some of the most relevant classes from the design package for this task.

- Cell: A cell corresponds to the leaf cell within the logical netlist EDIFCellInst and provides a mapping to a physical location BEL on the device. A cell can be created directly out of an EDIFCellInst to inherit all of its edif properties on the design level.
- Net: Represents the physical net to be routed (both inter-site and intra-site). When an Cell is created out of an EDIFCellInst, the Nets are automatically created out of its corresponding EDIFNets.
- SiteInst: An instance of a Site on the Device. Carries the mapping information between the BELs in a Site and the Cells assigned to them. Also keeps track of the intra-Site routing information within (Nets, SitePinInsts, SitePIPs, etc.).
- SitePIP: A Programmable Interconnect Point (PIP) in a Site. Represents the fuses in intra-Site routing BELs.
- SitePinInst: An instance of a SitePin on a Site. These objects serve as the interface between intra-Site routing and general inter-Site routing.

A SiteInst object can be created and its BELs populated with existing EDIFHierCellInst objects from the EDIFNetlist. A SiteInst is placed on a specific device Site upon its creation using the constructor in Listing 5. The packer therefore assigns an initial placement to every SiteInst before the simulated-annealing stage randomizes their positions. During this packing phase, we simply map design SiteInsts onto device Sites in coordinate order as they are generated (e.g., the first SiteInst onto SLICEL_X0Y0, the second onto SLICEL_X0Y1, and so on).

Listing 5 shows how to use the SiteInst object and how it ties Sites, Cells, BELs, and intra-Site Nets together. The super-specifics of the packing process via RapidWright is too involved for the scope of this paper, but just to give an impression of how the design classes are used, Listing 6, 7, 8, and 9 together show how CarryCellGroups are packed into SLICEL Sites in our working code.

Packing the other Cell structures into Sites follow a similar process. For each cell cluster, we create a SiteInst and populate its BELs with the corresponding Cells in the cluster and assign it a physical Site on the device. We further consolidate these SiteInsts into clusters or lists of SiteInsts for multi-Site structures like CarryCellGroup chains and DSP cascades. For each SiteInst, we assign it a physical Site on the device and route its internal intra-Site nets. The end result is a PackedDesign object which is a group of data structures solely around SiteInsts like shown in figure 28. At this point we no longer need to worry about individual BELs, Cells, or intra-Site Nets as they have now been packaged and taped-off into their respective SiteInsts. These SiteInst become the new atomic level components in the following Placement stage. This process is analogous to the containerization of loose goods in ocean freight forwarding.

Listing 5: SiteInst constructor and methods.

```
SiteInst Constructor:
    SiteInst(String name, Design design, SiteTypeEnum type, Site site)
Most relevant SiteInst Methods:
    createCell(EDIFHierCellInst inst, BEL bel) // Populating the SiteInst BELs with Cells
    unplace() // Unplacing the SiteInst from its current Site
    place(Site site) // Placing an unplaced SiteInst onto a Site
    routeSite() // Attempt to automatically route all intra-Site nets (manual intervention likely required)
    routeIntraSiteNet(Net net, BELPin src, BELPin snk) // Manually route an intra-Site net

Example:
    SiteInst si = new SiteInst("mySiteInst", design, SiteTypeEnum.SLICEL, device.getSite("SLICEL_X0Y1"));
    si.createCell(someFDRECell, si.getBEL("AFF"));
    si.routeSite();
    si.unplace();
    si.place(device.getSite("SLICEL_X15Y33"));
    // In Simulated Annealing, SiteInst objects will be unplaced() and placed() many times to converge to an optimal solution.
    // All Cell-BEL mapping and intra-Site routing is preserved when a SiteInst is moved.
```

Listing 6: Packing one CarryCellGroup into one SLICEL SiteInst.

```
// Names of the BELs in the 4 LUT-FF lanes:
protected String[] FF_BELS = new String[] { "AFF", "BFF", "CFF", "DFF" };
protected String[] LUT6_BELS = new String[] { "A6LUT", "B6LUT", "C6LUT", "D6LUT" };
// Pack one CarryCellGroup
private void packCarrySite(CarryCellGroup carryCellGroup, SiteInst si) {
    // Iterate through 4 LUT-FF Lanes
    for (int i = 0; i < 4; i++) {
        EDIFHierCellInst ff = carryCellGroup.ffs().get(i);
        if (ff != null)
            si.createCell(ff, si.getBEL(FF_BELS[i]));
        EDIFHierCellInst lut = carryCellGroup.luts().get(i);
        if (lut != null)
            si.createCell(lut, si.getBEL(LUT6_BELS[i]));
        // carry site LUTs MUST be placed on LUT6 BELs.
        // only LUT6/O6 can connect to CARRY4/S0
    }
    si.createCell(carryCellGroup.carry(), si.getBEL("CARRY4"));
    // default intrasite routing
    si.routeSite();
    // sometimes the default routeSite() is insufficient, so some manual intervention is required
    rerouteCarryNets(si);
    rerouteFFClkSrCeNets(si);
} // end placeCarrySite()
```

Listing 7: Packing `CarryCellGroups` into `SLICEL` `SiteInsts`.

```
// Pack all CarryCellGroup chains
private List<List<SiteInst>> packCarryChains(List<List<CarryCellGroup>> EDIFCarryChains)
        throws IOException {
    List<List<SiteInst>> siteInstChains = new ArrayList<>();
    writer.write("\n\nPacking carry chains... (" + EDIFCarryChains.size() + ")");
    for (List<CarryCellGroup> edifChain : EDIFCarryChains) {
        List<SiteInst> siteInstChain = new ArrayList<>();
        writer.write("\n\t\tChain Size: (" + edifChain.size() + "), Chain Anchor: "
                + edifChain.get(0).carry().getFullHierarchicalInstName());
        Site anchorSite = selectCarryAnchorSite(edifChain.size());
        SiteTypeEnum selectedSiteType = anchorSite.getSiteTypeEnum();
        for (int i = 0; i < edifChain.size(); i++) {
            Site site = (i == 0) ? anchorSite
                    : device.getSite("SLICE_X" + anchorSite.getInstanceX() + "Y" + (anchorSite.getInstanceY() + i));
            SiteInst si = new SiteInst(edifChain.get(i).carry().getFullHierarchicalInstName(), design,
                    selectedSiteType,
                    site);
            packCarrySite(edifChain.get(i), si);
            if (i == 0) { // additional routing logic for anchor site
                Net CINNet = si.getNetFromSiteWire("CIN");
                CINNet.removePin(si.getSitePinInst("CIN"));
                si.addSitePIP(si.getSitePIP("PRECYINIT", "0"));
            }
            occupiedSites.get(selectedSiteType).add(site);
            availableSites.get(selectedSiteType).remove(site);
            siteInstChain.add(si);
        }
        siteInstChains.add(siteInstChain);
    } // end for (List<EDIFCellInst> chain : EDIFCarryChains)
    return siteInstChains;
} // end packCarryChains()
```

Listing 8: Manually rerouting intra-Site nets in a `SLICEL` containing a `CARRY4` `Cell`

```
protected String[] FF_BELS = new String[] { "AFF", "BFF", "CFF", "DFF" };
private void rerouteCarryNets(SiteInst si) {
    // activate PIPs for CARRY4/COUT
    si.addSitePIP(si.getSitePIP("COUTUSED", "0"));
    // undo default CARRY4/DI nets
    SitePinInst AX = si.getSitePinInst("AX");
    if (AX != null)
        si.unrouteIntraSiteNet(AX.getBELPin(), si.getBELPin("ACY0", "AX"));
    SitePinInst DX = si.getSitePinInst("DX");
    if (DX != null)
        si.unrouteIntraSiteNet(DX.getBELPin(), si.getBELPin("DCY0", "DX"));
    // activate PIPs for CARRY4/DI pins
    si.addSitePIP(si.getSitePIP("DCY0", "DX"));
    si.addSitePIP(si.getSitePIP("CCY0", "CX"));
    si.addSitePIP(si.getSitePIP("BCY0", "BX"));
    si.addSitePIP(si.getSitePIP("ACY0", "AX"));
    // remove stray CARRY4/CO nets
    if (si.getNetFromSiteWire("CARRY4_CO2") != null)
        design.removeNet(si.getNetFromSiteWire("CARRY4_CO2"));
    if (si.getNetFromSiteWire("CARRY4_CO1") != null)
        design.removeNet(si.getNetFromSiteWire("CARRY4_CO1"));
    if (si.getNetFromSiteWire("CARRY4_CO0") != null)
        design.removeNet(si.getNetFromSiteWire("CARRY4_CO0"));
    // add default XOR PIPs for unused FFs
    for (String FF : FF_BELS)
        if (si.getCell(FF) == null)
            si.addSitePIP(si.getSitePIP(FF.charAt(0) + "OUTMUX", "XOR"));
} // end rerouteCarryNets()
```

Listing 9: Manually reroute intra-Site nets in a `SLICEL` containing FF `Cells`

```
private void rerouteFFClkSrCeNets(SiteInst si) {
    si.addSitePIP("CLKINV", "CLK");
    // activate PIPs for SR and CE pins
    Net SRNet = si.getNetFromSiteWire("SRUSEDMUX_OUT");
    Net CENet = si.getNetFromSiteWire("CEUSEDMUX_OUT");
    // if routeSite() default PIPs are incorrect, deactivate them then activate correct PIP
    if (SRNet != null) {
        if (SRNet.isGNDNet()) {
            if (si.getSitePIPStatus(si.getSitePIP("SRUSEDMUX", "IN")) == SitePIPStatus.ON)
                for (String FF : FF_BELS)
                    si.unrouteIntraSiteNet(si.getSitePinInst("SR").getBELPin(), si.getBELPin(FF, "SR"));
            si.addSitePIP(si.getSitePIP("SRUSEDMUX", "0"));
        } else {
            if (si.getSitePIPStatus(si.getSitePIP("SRUSEDMUX", "0")) == SitePIPStatus.ON)
                for (String FF : FF_BELS)
                    si.unrouteIntraSiteNet(si.getBELPin("SRUSEDGND", "0"), si.getBELPin(FF, "SR"));
            si.addSitePIP(si.getSitePIP("SRUSEDMUX", "IN"));
        }
    }
    if (CENet != null) {
        if (CENet.isVCCNet()) {
            if (si.getSitePIPStatus(si.getSitePIP("CEUSEDMUX", "IN")) == SitePIPStatus.ON)
                for (String FF : FF_BELS)
                    si.unrouteIntraSiteNet(si.getSitePinInst("CE").getBELPin(), si.getBELPin(FF, "CE"));
            si.addSitePIP(si.getSitePIP("CEUSEDMUX", "1"));
        } else {
            if (si.getSitePIPStatus(si.getSitePIP("CEUSEDMUX", "1")) == SitePIPStatus.ON)
                for (String FF : FF_BELS)
                    si.unrouteIntraSiteNet(si.getBELPin("CEUSEDGND", "1"), si.getBELPin(FF, "CE"));
            si.addSitePIP(si.getSitePIP("CEUSEDMUX", "IN"));
        }
    }
} // end rerouteFFSrCeNets()
```

10 Placement

So far we have only dealt with technology problems where we consider the problem space and adress the constraints of the hardware architecture. For every constraint, we devise a solution to fully address it or a simplification that sacrifices some optimization but reduces the complexity of the problem space.

Now that we have dealt with the bulk of these constraints and squared them away into self-contained `SiteInsts`, we can now more easily apply some general optimization algorithms to finally address our optimization objective, which is to place these `SiteInst` objects onto the device while minimizing wirelength.

10.1 Simulated Annealing

In this paper we implement and present a basic Simulated Annealing (SA) placer. As mentioned in the Introduction, SA is a metaheuristic that approximates a global optimum in a large optimization search space. SA does not guarantee a globally optimal solution, but provides solutions that are often "good enough", especially when finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time. SA is typically employed in discrete combinatorial optimization problems like the travelling salesman problem or job-shop scheduling. It is directly inspired by Annealing in metallurgy, which is the physical process of heating a metal above its crystallization temperature, allowing its atoms to migrate through the crystal lattice, and slowly cooling it to allow its atoms to recrystallize into a more desirable structure, such as one that minimizes structural defects.

The state s of the system and the function $E(s)$ to be minimized is analogous to the internal energy of the system. In metallurgic Annealing, the state s represents the position of every atom in the metal object at any given snapshot in time, while $E(s)$ represents the defectiveness in the metal's structure however that may be quantified. In the context of our SA placer, the state s represents the placement of every `SiteInst` on the `device` at any given iteration, while $E(s)$ represents the total wirelength (HPWL) of `Nets` between all `SiteInsts` of the current placement. The goal of SA is to bring the system from some initial state to a state with lower energy until some stopping criterion is met, whether that be when a computational budget is exceeded, an energy budget is met, or until the energy's rate of change approaches zero.

At each step or iteration, the algorithm considers some neighboring state s^*. If s^* has a lower energy than s, then the state transition from s to s^* is accepted outright. If s^* has a higher energy, then it can probabilistically be accepted depending on the current global temperature. The higher the temperature, the higher the likelihood of accepting an energy-increasing move. The global temperature starts at some positive amount then gradually decreases to zero according to a cooling schedule. This cooling schedule can be linear, geometric, logarithmic, or some piecewise combination. Its parameters can be fine tuned through empirical experimentation.

Shown in Listing 10 is a simplified pseudocode for the outer-most loop of our placer's SA algorithm. First, we randomly place all `SiteInsts` across the `device`. Then, we initialize a simple geometric cooling schedule:

$$T_{n+1} = \alpha T_n \tag{1}$$

with initial temperature $T_0 = 10000$, cooling rate $\alpha = 0.98$, and $n = \{0, 1, 2, ..., 300\}$ passes.

Then, we enter the outer-most loop which terminates when the number of passes exceeds our computational budget, in this case, 300 passes. In each pass, we `moveAll()` the design, then update the global temperature.

In each `moveAll()`, we iterate through four mutually exclusive groups of placement objects: `DSPSiteInstCascades`, `CARRYSiteInstChains`, `RAMSiteInsts`, and `CLBSiteInsts` (single SLICEs that do not belong to CARRY chains). For each placement object in the pass, we propose a move to a new placement on the device. If the proposed `Site` is already occupied by a resident or existing `SiteInst`, we evaluate a swap in their placements. Each move or swap is evaluated by calculating the HPWL cost of the placement before and after the proposed movement. If the design has for example, 50 `DSPSiteInstCascades`, 100 `CARRYSiteInstChains`, 75 `RAMSiteInsts`, and 250 `CLBSiteInsts`, and we set a budget of 300 passses, then the algorithm will consider up to $300 * (50 + 100 + 75 + 250) = 142500$ state transitions.

In SA, proposed movements are selected randomly, then accepted or rejected based on legality and cost. Figure 31 shows how a swap proposal between singular `CLBSiteInsts` or between `RAMSiteInsts` is evaluated, while Listing 11 shows the corresponding Java code. On the other hand, the movement of `SiteInst` chains is considerably more complex because there are more placement constraints to consider, especially when swapping the positions between multiple chains. Figure 32 shows a demonstration of how two `DSPSiteInstCascades` or two `CARRYSiteInstChains` can be swapped while Listing 13 shows the corresponding pseudocode.

In more sophisticated placers, the proposed movement can also be selected by finding the centroid between `SiteInsts` that share a net with the current object, or even a hybrid of random and centroid selection. Moves that actively attempt to predict a better placement are often referred to as "directed" moves in literature. Movement approaches that take inspiration from physical interactions like gravity or electrostatics are often referred to as "force directed" moves.

If the move reduces the HPWL cost, then the movement is accepted outright. If the move increases cost, then it can be accepted by chance if the global temperature is high enough to permit the hill-climb. The temperature decreases with each pass until reaching zero, at which point, the algorithm makes exclusively energy-decreasing moves, effectively reducing into a greedy algorithm. The hope is that by this point, the placement will crystallize into a global optimum of minimum total HPWL.

Listing 10: SA pseudocode: outer loop

```
public void placeDesign(PackedDesign packedDesign) {
    randomInitialPlacement(packedDesign);
    // init_temp=10000, alpha=0.98, max_passes=300
    initCoolingSchedule(10000, 0.98, 300)
    int passes = 0;
    while (passes < 300) {
        updateTemperature(passes);
        move(packedDesign)
        passes++;
    }
}

private void moveAll(PackedDesign packedDesign) {
    moveSiteChains(packedDesign.DSPSiteInstCascades);
    moveSiteChains(packedDesign.CARRYSiteInstChains);
    moveSingleSite(packedDesign.RAMSiteInsts);
    moveSingleSite(packedDesign.CLBSiteInsts);
}
```

Listing 11: Single Site Movement

```
protected void moveSingleSite(List<SiteInst> sites) {
for (SiteInst si : sites) {
    SiteTypeEnum ste = si.getSiteTypeEnum();
    List<Site> homeConns = findConnectedSites(si, null);
    Site homeSite = si.getSite();
    Site awaySite = proposeSite(si, homeConns, true);
    SiteInst awaySi = occupiedSites.get(ste).get(awaySite);
    double oldCost = 0;
    double newCost = 0;
    if (awaySi != null) {
        List<Site> awayConns = findConnectedSites(awaySi,
     null);
        oldCost += evaluateSite(homeConns, homeSite);
        oldCost += evaluateSite(awayConns, awaySite);
        newCost += evaluateSite(homeConns, awaySite);
        newCost += evaluateSite(awayConns, homeSite);
    } else {
        oldCost += evaluateSite(homeConns, homeSite);
        newCost += evaluateSite(homeConns, awaySite);
    }
    if (evaluateMoveAcceptance(oldCost, newCost)) {
        if (awaySi != null) {
            unplaceSiteInst(si);
            unplaceSiteInst(awaySi);
            placeSiteInst(si, awaySite);
            placeSiteInst(awaySi, homeSite);
        } else {
            unplaceSiteInst(si);
            placeSiteInst(si, awaySite);
        }
    }
}
} // end randomMoveSingleSite()
```

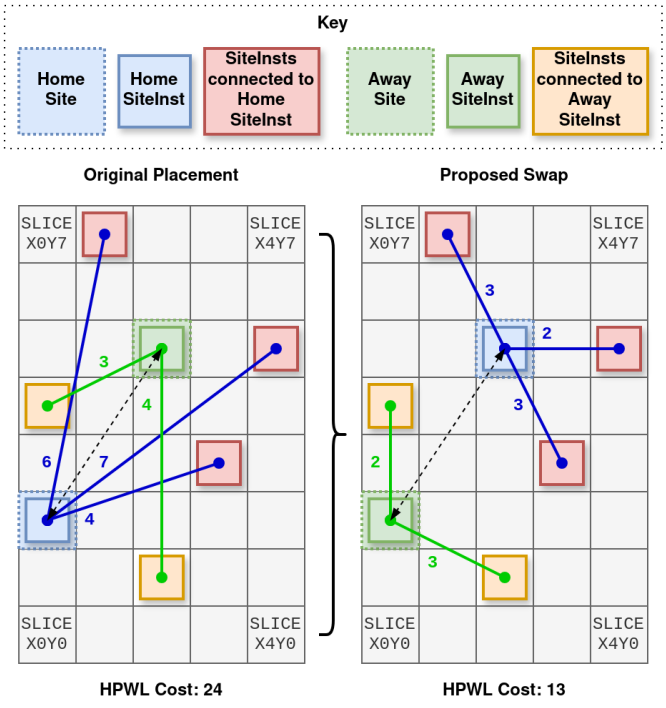


Figure 31: Single Site Swap Proposal

Listing 12: Cooling Schedule and Move Acceptance with Temperature

```
protected List<Double> coolingSchedule;
public void initCoolingSchedule(double initialTemp, double
     alpha, int movesLimit) {
    double currentTemp = initialTemp;
    for (int i = 0; i < movesLimit; i++) {
        this.coolingSchedule.add(currentTemp);
        currentTemp *= alpha; // geometric cooling
    }
}

protected boolean evaluateMoveAcceptance(double oldCost,
     double newCost) {
    // if the new cost is lower, accept it outright
    if (newCost < oldCost)
        return true;
    // otherwise, evaluate probability to accept higher cost
    double delta = newCost - oldCost;
    double acceptanceProbability =
        Math.exp(-delta / this.currentTemp);
    return Math.random() < acceptanceProbability;
}
```

Listing 13: Chain Swapping Pseudocode

```
protected void moveSiteChains(List<List<SiteInst>> chains) {
for (List<SiteInst> currentChain : chains) {
    int chainSize = currentChain.size();

    // Step 1: Identify home window for this chain
    Site homeAnchor = currentChain.get(0).getSite();
    List<Site> homeWindow = getSitesInWindow(homeAnchor,
     chainSize);

    // Step 2: Select a candidate away anchor
    Site awayAnchor = proposeAnchorSite(currentChain,
     homeWindow, true);

    // Step 3: Determine away window based on awayAnchor and
     chainSize
    List<Site> awayWindow = getSitesInWindow(awayAnchor,
     chainSize);

    // Step 4: Find any resident SiteInst chains in the away
     window
    List<List<SiteInst>> residentChainsInAway =
        collectChainsInWindow(siteType, awayWindow);

    // Step 5: If any resident chains overlap with the away
     window, extend the away window to fully accomodate them
    if (!residentChainsInAway.isEmpty()) {
        awayWindow = extendWindowToIncludeChains(awayWindow,
     residentChainsInAway);
    }

    // Step 6: Map the (possibly extended) away window back
     onto the original region so that the tail of that window
     coincides with the tail of the current chain
    List<Site> candidateHomeWindow = mapAwayToHomeWindow(
        homeAnchor, awayWindow, chainSize);

    // Step 7: While the candidate home window still overlaps
     resident chains, shift upward
    int shifts = 0;
    while (windowHasOverlap(candidateHomeWindow)) {
        candidateHomeWindow =
     shiftWindowUp(candidateHomeWindow);
        shifts++;
        if (shifts > (homeWindow.size() - chainSize)) {
            // Reject this swap attempt.
            // Move on to the next chain.
            continue;
        }
    }

    // Step 8: Compute cost of the original placement in home
     window
    double oldCost = evaluateWindowCost(homeWindow);

    // Step 9: Compute cost of the proposed swap placement
    double newCost = evaluateWindowCostForSwap(
        homeWindow, awayWindow, currentChain);

    // Step 10: Decide whether to accept the move based on
     oldCost, newCost, and temperature
    if (evaluateMoveAcceptance(oldCost, newCost, currentTemp))
     {
        // Step 11: Perform an element-wise swap of SiteInsts
     between homeWindow and awayWindow
        swapChainsBetweenWindows(homeWindow, awayWindow);
    }
}
}
```

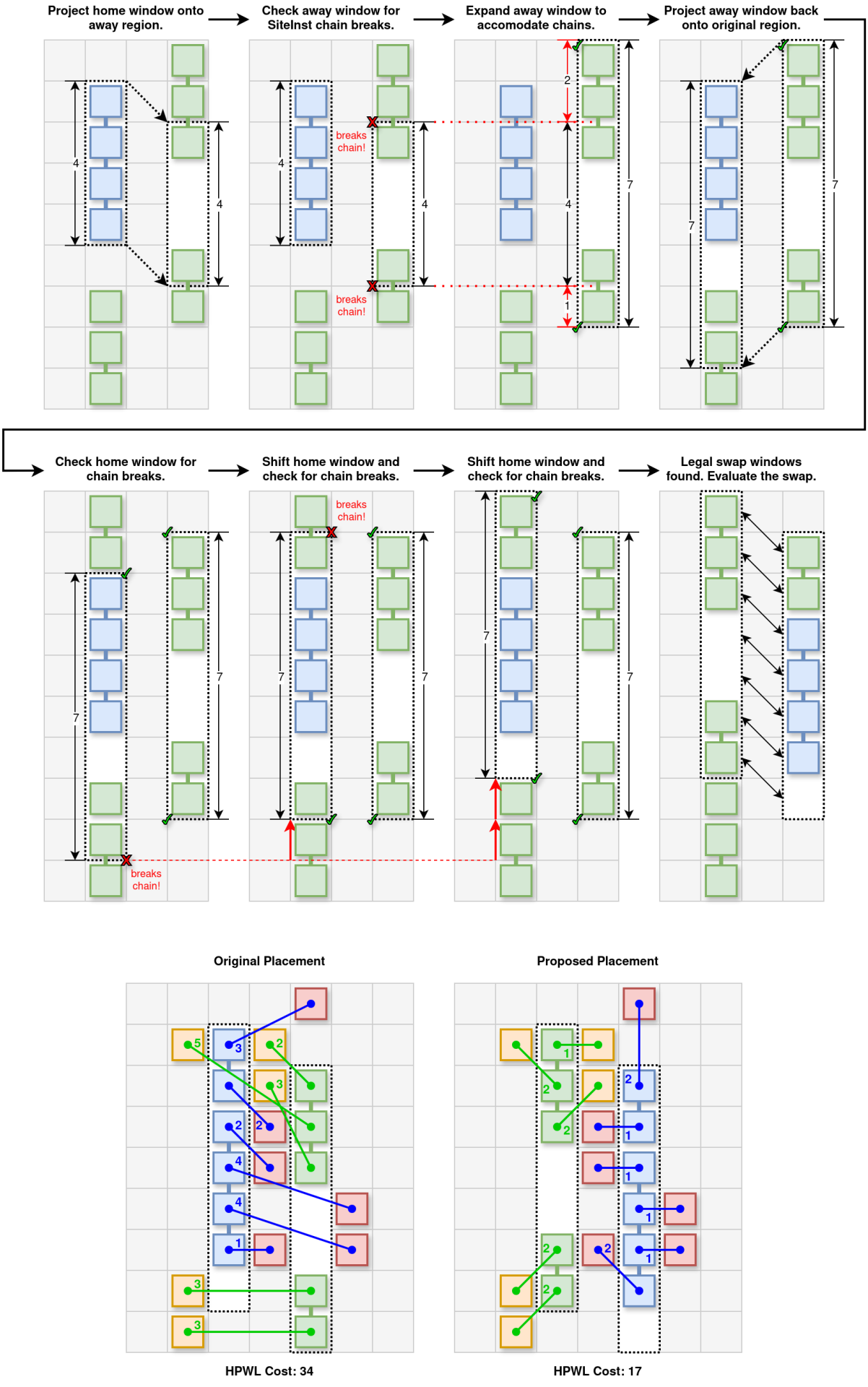


Figure 32: Site Chain Swap Proposal

11 Placement Results

11.1 Greedy Algorithm (Undirected/Random Moves)

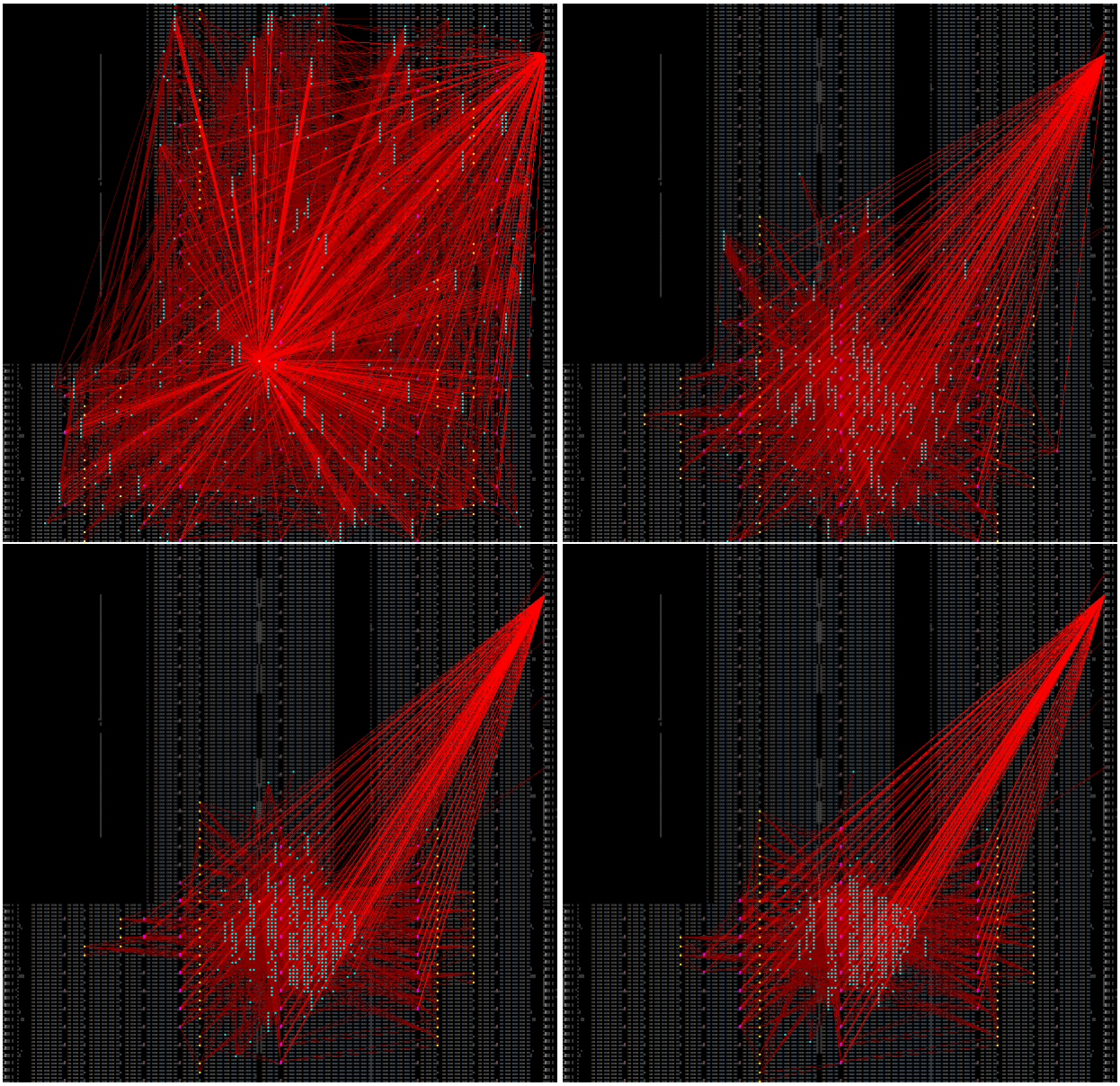


Figure 33: Snapshots of placement of a 2048th order FIR Filter on a xc7z020 FPGA. **From left to right: (1)** Initial random placement, **(2)** After 10 passes, **(3)** After 100 passes, **(4)** Final placement after 300 passes.

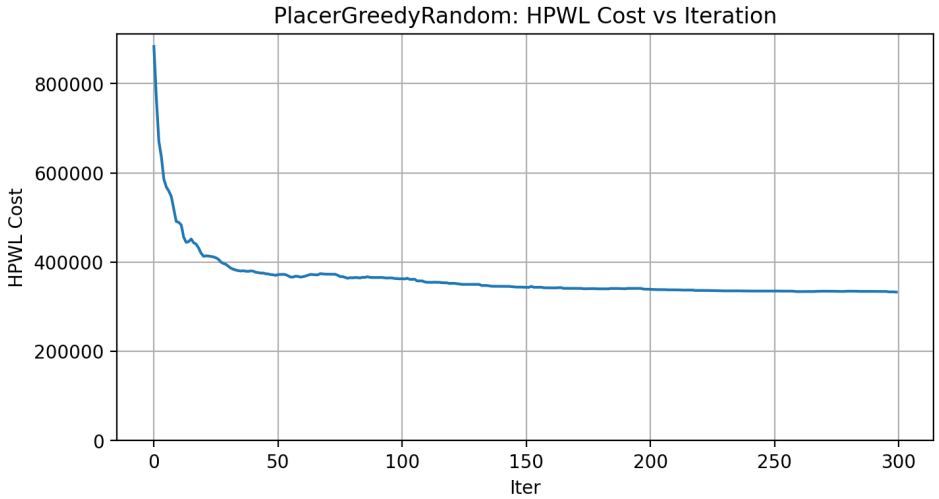


Figure 34: Total HPWL Cost vs number of passes. Final cost: 333013.

11.2 Greedy Algorithm (Directed/Midpoint Moves)

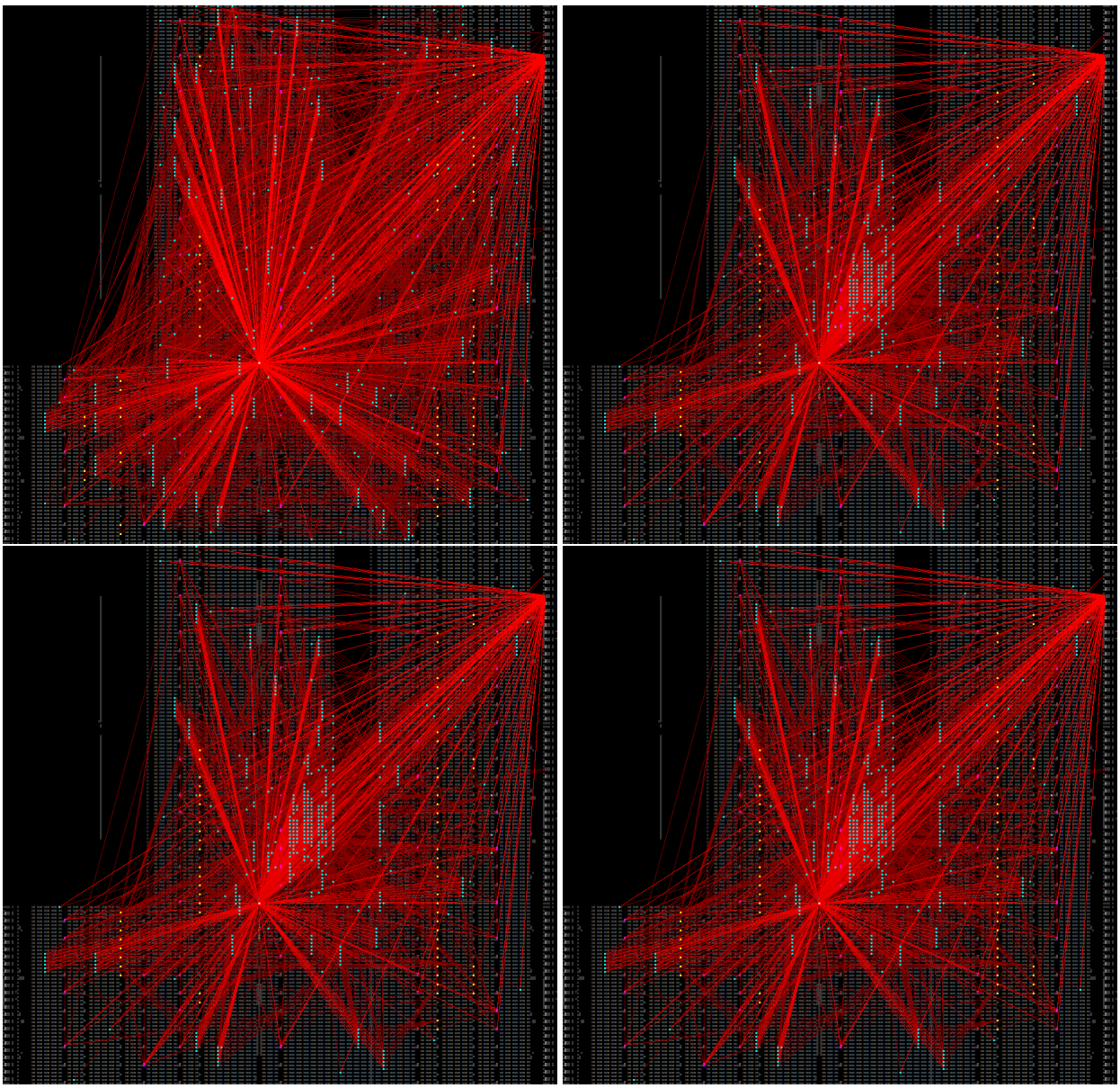


Figure 35: Snapshots of placement of a 2048th order FIR Filter on a xc7z020 FPGA. **From left to right: (1)** Initial random placement, **(2)** After 10 passes, **(3)** After 100 passes, **(4)** Final placement after 300 passes.

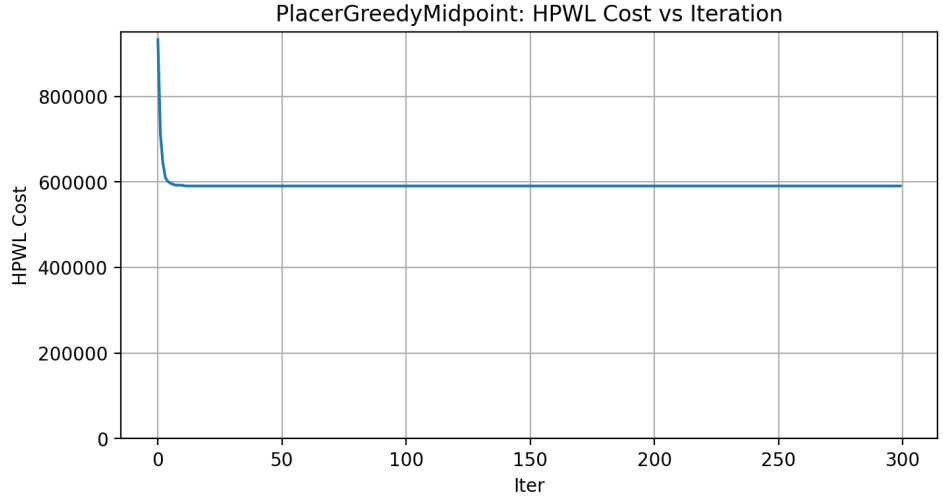


Figure 36: Total HPWL Cost vs number of passes. Final cost: 590578.

11.3 Simulated Annealing (Undirected/Random Moves)

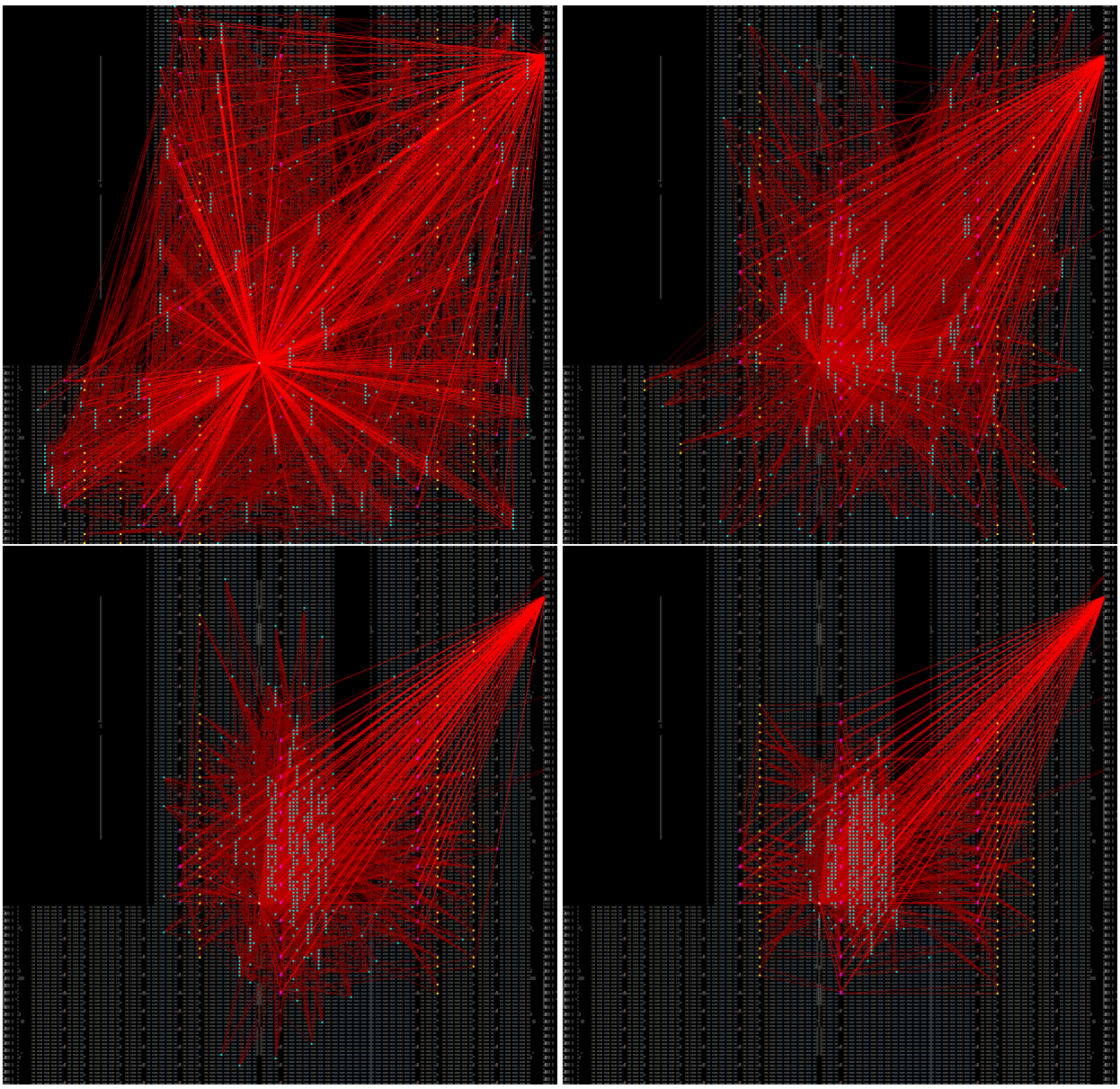


Figure 37: Snapshots of placement of a 2048th order FIR Filter on a xc7z020 FPGA. **From left to right: (1)** Initial random placement, **(2)** After 10 passes, **(3)** After 100 passes, **(4)** Final placement after 300 passes.

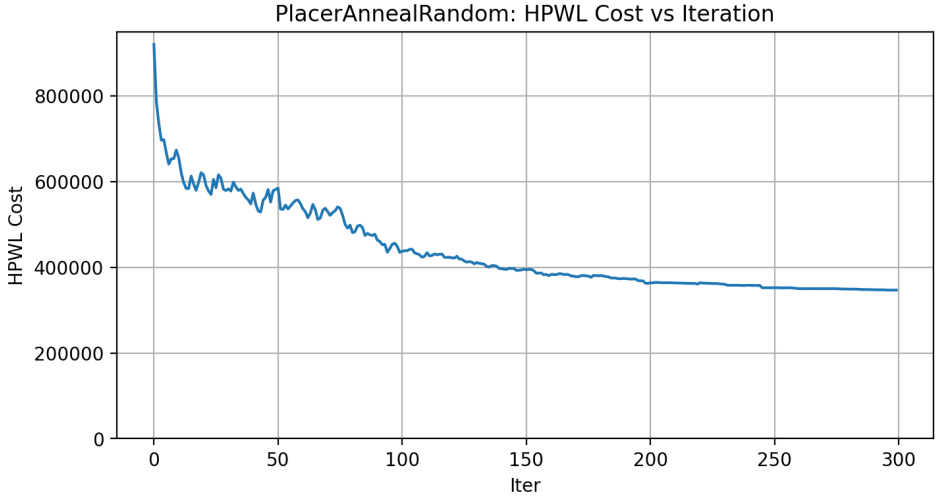


Figure 38: Total HPWL Cost vs number of passes. Final cost: 346676.

11.4 Simulated Annealing (Directed/Midpoint Moves)

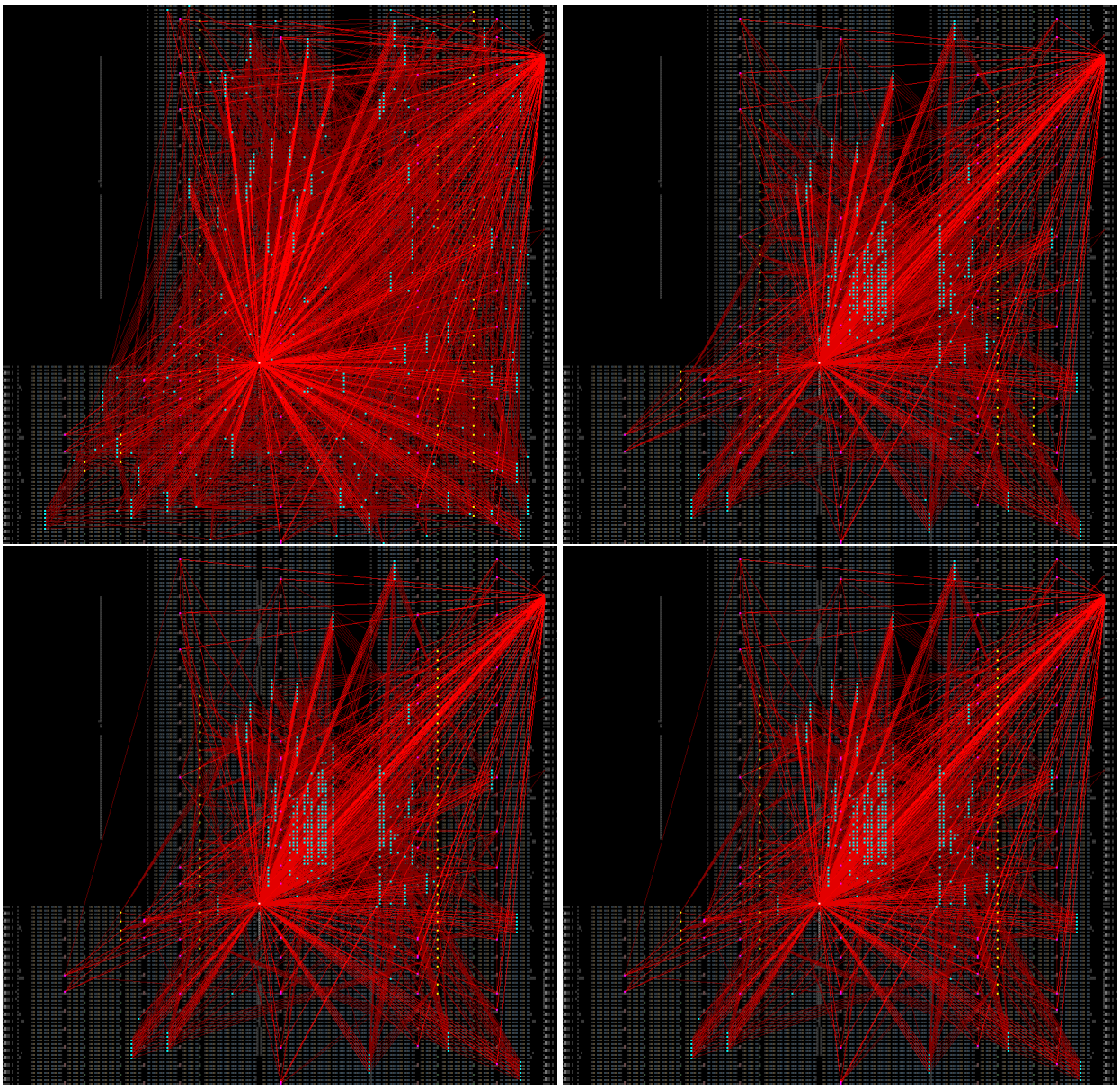


Figure 39: Snapshots of placement of a 2048th order FIR Filter on a xc7z020 FPGA. **From left to right: (1)** Initial random placement, **(2)** After 10 passes, **(3)** After 100 passes, **(4)** Final placement after 300 passes.

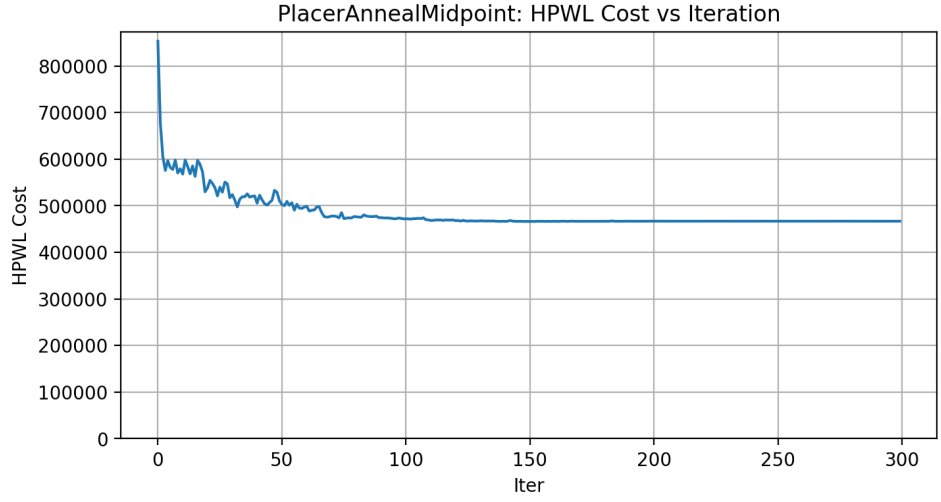


Figure 40: Total HPWL Cost vs number of passes. Final cost: 466855.

11.5 Simulated Annealing (Hybrid: Random + Midpoint)

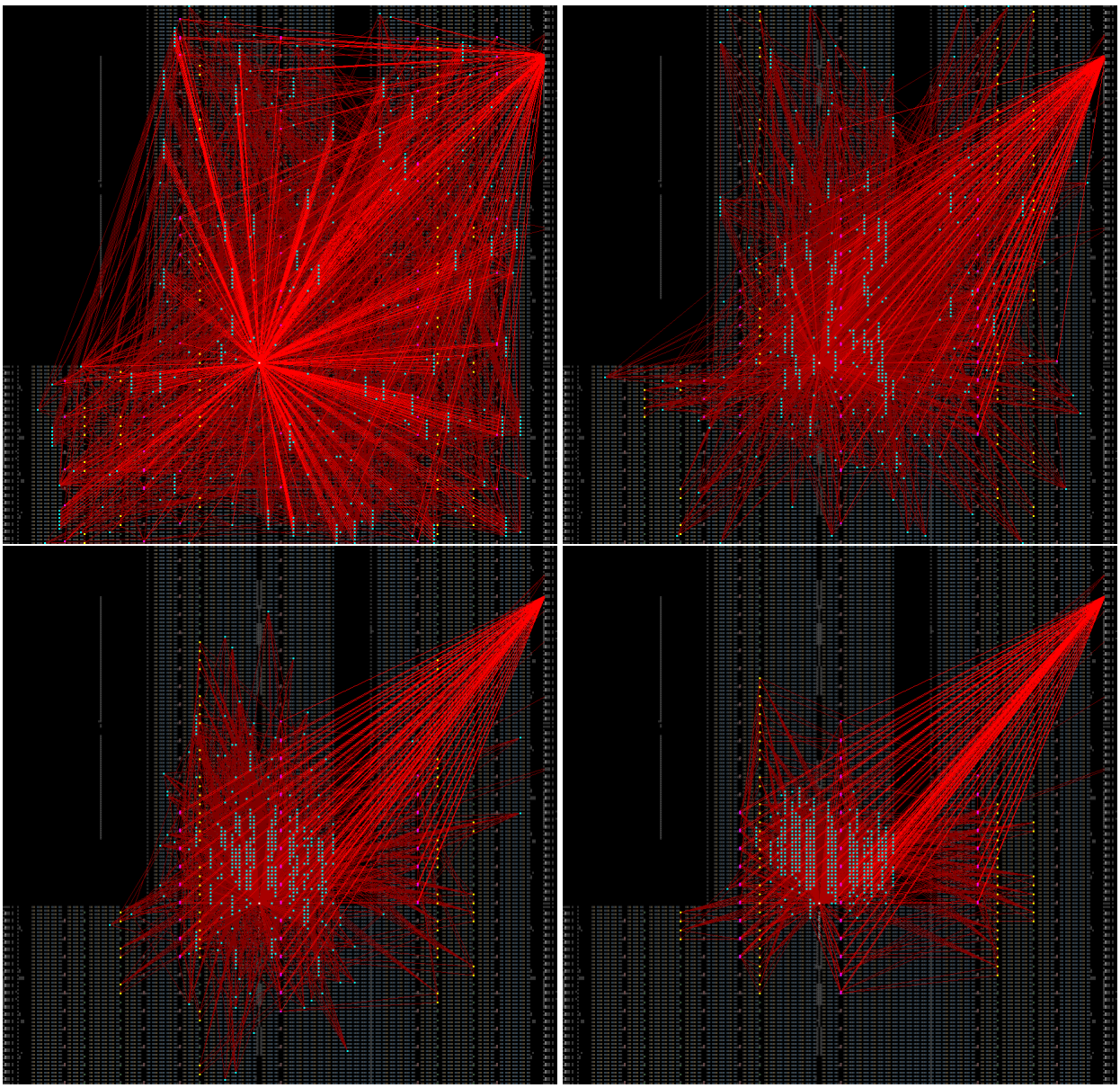


Figure 41: Snapshots of placement of a 2048th order FIR Filter on a xc7z020 FPGA. **From left to right: (1)** Initial random placement, **(2)** After 10 passes, **(3)** After 100 passes, **(4)** Final placement after 300 passes.

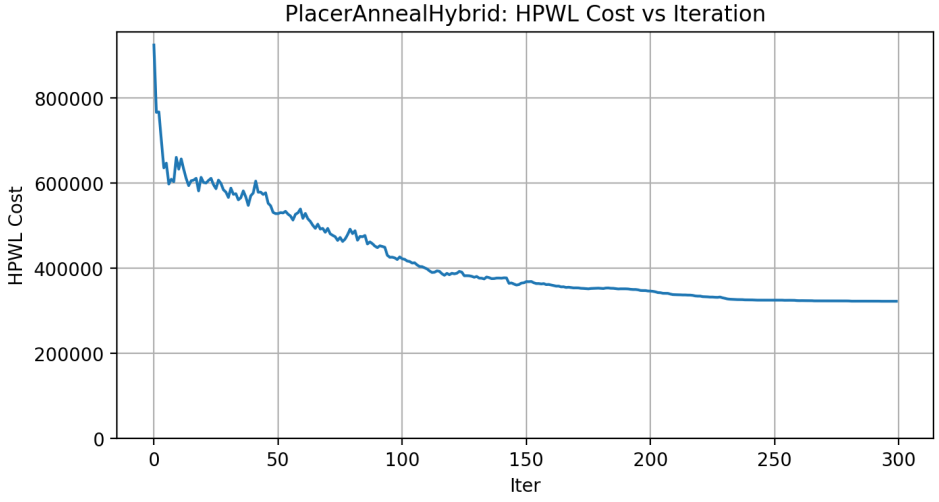


Figure 42: Total HPWL Cost vs number of passes. Final cost: 322933.