MASTER THESIS

Verilog RTL Regularity Analysis & Effect on Structured Datapath Placement

Student:
Maria Pantazi-Kypraiou
mpantazi-@uth.gr

Supervisor:
Christos Sotiriou
chsotiriou@e-ce.uth.gr
Committee:
Georgios Stamoulis
georges@e-ce.uth.gr
Fotios Plessas
fplessas@e-ce.uth.gr

A thesis submitted in fulfillment of the requirements
for the degree of Master
in the
Circuits & Systems Laboratory (CASlab)
Department of Electrical and Computer Engineering

March 7, 2023

Abstract

Several combinational logic architectures (i.e. adders, neural networks and more) present high structural uniformality and regularity, due to similar or identical parallel functional data processes. In this work, a verilog RTL techniques flow, of three steps, is proposed to help increase regularity, a method of extracting regularity using login cones for designs with or without logic depth (i.e. adders, MUXs) when RTL is not available, or the designer does not want to change it for better optimizations during synthesis and finally, a bit-slicing, or not, placement flow is proposed, custom or automated within some contexts, to optimize HPWL. The case studies are a register file, a CSA booth multiplier and a DLX execute unit. Firstly, for each design the regularity extractions is demonstrated and after that, a hierarchical datapath flow, performing an SDP-like (Structure DataPath) placement.

Περίληψη

Πολλές αρχιτεκτονικές συνδυαστικής λογικής (όπως προσθετές και νευρωνικά δίκτυα) παρουσιάζουν υψηλή δομική ομοιομορφία και κανονικότητα, λόγω παράλληλων λειτουργικών διεργασιών δεδομένων. Σε αυτήν την εργασία, προτείνεται μία ροή τριών τεχνικών που εφαρμόζονται σε επίπεδο γλώσσας περιγραφής υλικού, για την αύξηση της ομοιομορφίας, μια μέθοδος εξαγωγής ομοιομορφίας χρησιμοποιώντας λογικούς κόνους για κυκλώματα με ή χωρίς βάθος λογικής (όπως προσθετές και πολυπλέκτες) όταν η περιγραφή σε κώδικα δεν είναι διαθέσιμη ή ο σχεδιαστής δεν επιθυμεί να την αλλάξει για βελτιστοποίηση κατά τη σύνθεση και τέλος προτείνεται μια ροή τοποθέτησης με το χέρι ή αυτοματοποιημένη σε ορισμένα πλαίσια, για τη βελτιστοποίηση του συνολικού μήκους των καλωδίων. Οι μελέτες περιλαμβάνουν ένα αρχείο καταχώρησης(register file), έναν πολλαπλασιαστή Booth με CSAs και μια μονάδα εκτέλεσης DLX. Αρχικά, για κάθε κύκλωμα, επιδεικνύεται η εξαγωγή ομοιομορφίας και στη συνέχεια η ιεραρχική ροή δομημένης τοποθέτησης.

Acknowledgements

I would like to thank my thesis supervisor Prof. Christos Sotiriou, who was always available and willing to help whenever I ran into trouble or had a question about my research and guided me in the right direction whenever he thought I needed it. I would also like to thank Prof. Georgios Stamoulis and Prof. Fotios Plessas for the confidence they showed in my face.

I would like also to thank the members of the CASlab, who created a friendly environment, and especially Dimitrios Valiantzas and Nikolaos Sketopoulos, who helped me in my thesis research.

Finally, I would like to express my acknowledgements to my family and my friends supporting me throughout my years of study at the University of Thessaly.

Maria Pantazi-Kypraiou
Volos, 2023

Verilog RTL Regularity Analysis & Effect on Structured Datapath Placement

Maria Pantazi-Kypraiou
mpantazi-@uth.gr

Ανάλυση Ομοιομορφίας σε Επίπεδο Γλώσσας Περιγραφής Υλικού και η Επίδραση της στην Δομημένη Τοποθέτηση

Μαρία Πανταζή-Κυπραίου
mpantazi-@uth.gr

Contents

List of Figures

Listings

Chapter 1

Introduction

Combinational logic architectures, such as adders and neural networks, often exhibit high structural uniformity and regularity due to their parallel functional data processes. This regularity can be leveraged to improve Power, Performance, and Area (PPA) results in the design process. By taking advantage of the regularity of these architectures, designers can develop custom placements that optimize the layout and routing of the design, reducing the overall wirelength, represented as Half Perimeter Wirelength (HPWL), area and power consumption. This approach can lead to improved performance and efficiency in the design process, as well as reduced manufacturing costs.

The placement step in the design flow of Application-Specific Integrated Circuits (ASICs) involves the physical placement of logic gates and other circuit components on a chip. To optimize the placement step for specific design architectures, designers can implement an optimal custom regular placement strategy. This involves designing custom placements that take into account the regularity and uniformity of the architecture, which can lead to reduced half-perimeter wirelength (HPWL), area, and power consumption. Once the custom placements have been developed, a macro-like placement strategy can be used to place the optimal custom-designed blocks in the layout. This strategy involves grouping similar blocks together and placing them in predefined areas of the layout to achieve greater regularity and uniformity. The use of a macro-like placement strategy can further improve PPA results, as it can help to reduce routing congestion and improve overall timing and power efficiency. By combining optimal custom regular placement with a macro-like placement strategy, designers can achieve better PPA results and improve the overall quality of the ASIC design.

Below, in fig. 1.1, the motivation is shown. On the left there is the flat placement of a 32-bit register file and on the right its SDP version. The design on the right has clearly less area and less HPWL, which will be shown later on the results.

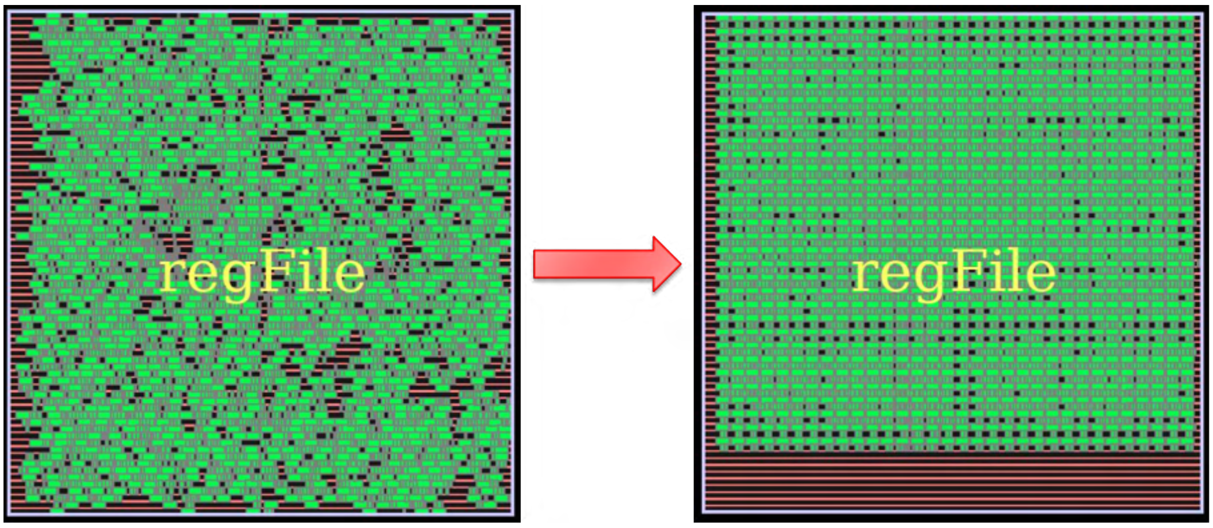


Figure 1.1: Motivation

Chapter 2

Theoretical Background

2.1 EDA & Physical Design Background

2.1.1 Introduction to EDA

An integrated circuit (IC) refers to a collection of electronic circuits that are situated on a diminutive flat semiconductor material, often composed of silicon. ICs have gained widespread use in electronic devices and have been instrumental in transforming the electronics industry. The current-day landscape of modern societies is marked by the ubiquitous presence of IC-powered devices, ranging from computers to mobile phones and household appliances. This is primarily attributed to the compact size and economical nature of ICs, including advanced computer processors and microcontrollers.

Back in the history, IC's were mainly designed by hand. However, with the rapid growth of the electronics industry in commonplace applications such as televisions, cars, cellphones etc and the increased performance demands, manual design has become impossible.

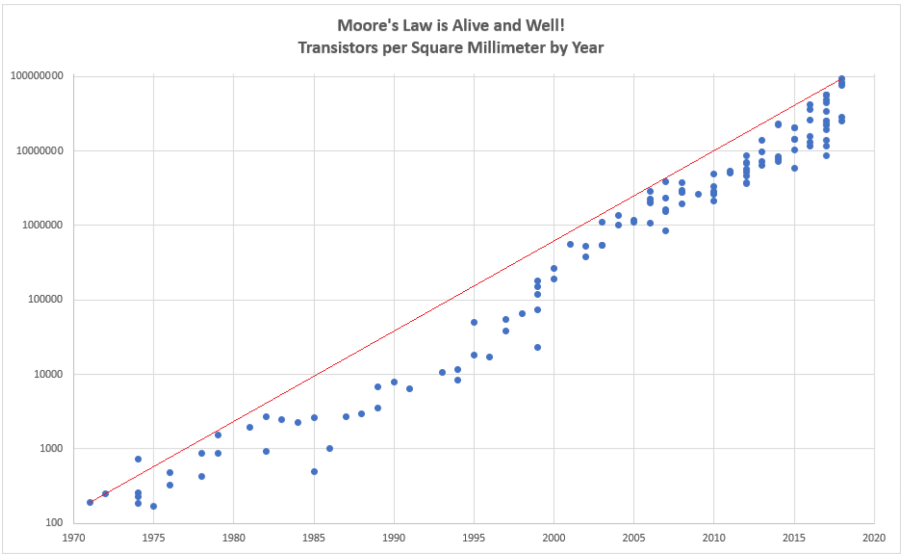


Figure 2.1: Moore's Law from 1970 till 2020

According to Moore's Law, shown in fig. 2.1 above, the number of transistors doubles every approximately two years, so the designing of very large-scale integrated (VLSI) circuits becomes more and more complex. Electronic Design Automation (EDA) industry has come to the rescue, developing software tools to aid the engineers design complex electronic systems more efficiently.

EDA tools automate the design process and enable engineers to create, simulate, and verify electronic systems with high efficiency and accuracy. EDA encompasses a broad range of applications, from schematic capture and layout to simulation and analysis. These tools are essential for designing modern electronics, such as microprocessors, memory devices, and communication systems, which require precise timing, low power consumption, and high reliability.

2.1.2 Physical Design

Physical design is the process of transforming a high-level description of a digital system into a detailed description of the physical components that will implement that system. It involves selecting the specific hardware components and determining how they will be interconnected to create the desired system.

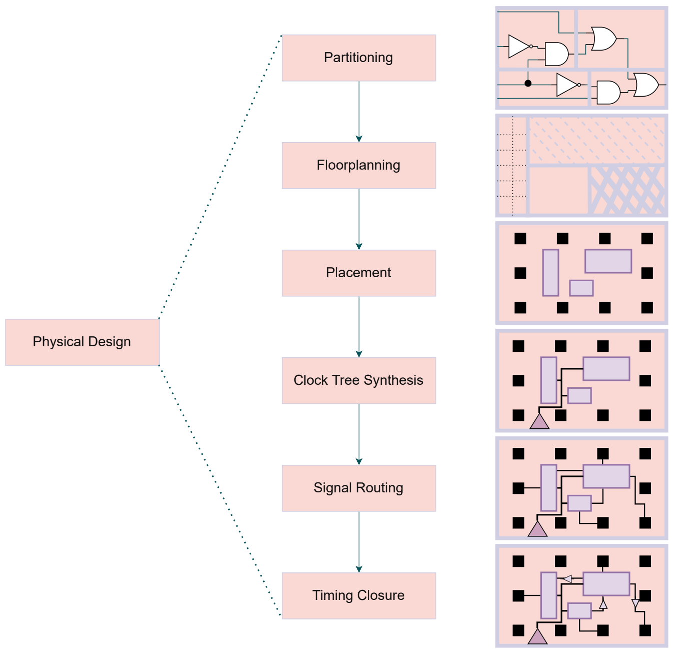


Figure 2.2: Physical Design Flow

As shown in fig. 2.2, physical design consists of a number of steps, including floorplanning, placement, clock tree synthesis, and routing. It starts from the netlist, a synthesized Register Transfer Level (RTL), and passes through floorplanning, the first major step. Floorplanning includes the identification of structures that should be placed near others, depending on area restrictions, speed or constraints required by components. Floorplanning is followed by partitioning, which involves dividing the system into smaller modules or subsystems that can be designed and implemented separately. The next step, and the one that will be discussed further below, is placement. During this step, each component's location on the Printed Circuit Board (PCB) is determined, as well as the routing of the connections between them. Finally, we have clock tree synthesis and routing. During the first, buffers or inverters are inserted in a way that the clock is distributed evenly to sequential elements in the design, minimizing skew and latency, while during the second, the paths of interconnects are determined, including standard cell and macro pins. The final output of the physical design process is typically GDSII, a data format representing layout information.

2.1.3 Placement

In digital circuit design, placement is the crucial step of positioning various components on a chip or a PCB. Its primary objective is to minimize wirelength, lower manufacturing costs and reduce power consumption, as such, it directly influences the efficiency and performance of a digital system. The placement process involves three stages, namely global placement, legalization, and detailed placement. These steps aim to ensure optimal component positioning and wiring density to enhance the overall design quality.

Global Placement (GP)

During global placement, the entire circuit is divided into smaller blocks or modules, and the positions of these blocks are optimized to achieve the desired performance and efficiency. The goal of global placement is to minimize the overall area of the circuit and reduce the length of the connections between the components.

Rather than paying attention to the placement grid rows, GP focuses on finding the optimal coordinates for the cells in order to minimize wirelength, while maintaining some density specifications. After this step, an approximation of gate's positions is determined, but cells may have overlaps with one another, making the next placement step (Legalization) essential.

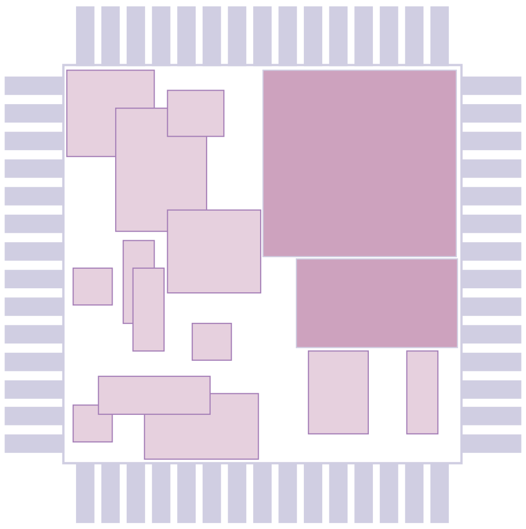


Figure 2.3: Global Placement

Legalization (LG)

Legalization involves adjusting the positions of the components to the placement manufacturing grid rows in order to meet the constraints and rules set by the IC fabrication process.

During the legalization process, modifications are made to the positions of components to ensure compliance with minimum feature sizes and spacing requirements. This may require the movement of components or the addition of dummy components to occupy any unutilized spaces. Essentially, the tool relocates cells to conform to legal positions on the placement grid and eliminates any overlaps between them. These slight modifications to cell positions can result in changes in the lengths of wire connections, which may give rise to new timing issues. However, such violations can typically be rectified through incremental optimization techniques, such as adjusting the sizes of the driving cells, or by carrying out detailed placement.

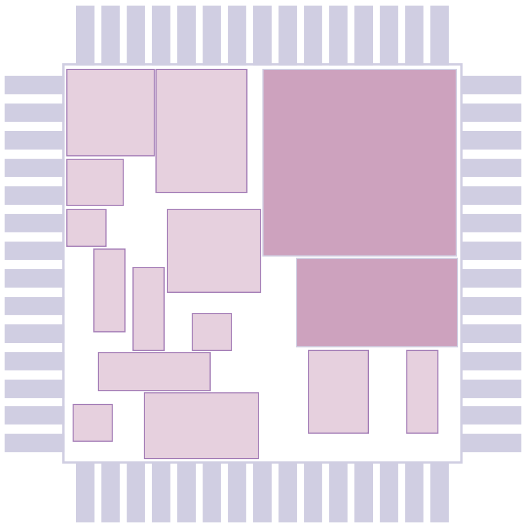


Figure 2.4: Legalisation

Detailed Placement

After obtaining a legalized placement solution, the next step is to carry out detailed placement to further enhance wirelength or other objectives by locally rearranging standard cells while ensuring that the placement remains legal. Detailed placement presents a significant opportunity for improving wirelength in a legalized global placement solution for several reasons. Firstly, the wirelength models used in global placement are typically inaccurate, such as cut cost, quadratic wirelength with clique net model, or log-sum-exponential function. Secondly, global placement algorithms often place cells into subregions with little regard for their position within that region. Finally, during legalization, wirelength may be adversely affected by perturbations. Therefore, detailed placement can help rectify any issues that arise and achieve better wirelength optimization.

2.2 Structured Datapath Background

Structured Datapath process consists of two steps, **regularity extraction** and **structured datapath placement**. Datapath is implemented in bit-slice structures to manipulate multiple bits of data simultaneously. The structures could also be made of several functional stages, that is multiplexers, arithmetic logic units (ALUs) and especially registers. Registers usually pipeline a basic datapath to reduce critical path issues and increase the operating frequency. Traditionally, datapath design was a process that could be done only manually, consumed a lot of time and a significant amount of human effort.

For **regularity extraction**, a set of structures that is repeated many times in a circuit netlist has to be identified. The techniques that exist in the literature for regularity extraction are numerous and they are further discussed in section 3.

Structured datapath placement is a technique used in the physical design of digital systems to optimize the layout of the system on a printed circuit board (PCB). The goal of structured datapath placement is to reduce the routing congestion and improve the performance of the system by organizing the components in a structured way. It is often used in the design of systems that have numerous components which need to be interconnected, such as processors.

In structured datapath placement, the components are first grouped into smaller modules or subsystems, each of which performs a specific function. They are, then, arranged in a pattern, such as a linear array or a two-dimensional grid. More efficient routing of signals between the components is allowed, as the connections follow a predictable pattern.

Chapter 3

Regularity Extraction & Measurements

In a netlist, regularity can be extracted by identifying patterns in the components and connections. For example, if a particular circuit element appears multiple times in the netlist, it could be considered a regular element. Similarly, if there are multiple connections with the same components, they could be considered a regular pattern.

This can be done by using techniques such as graph theory, which can be used to analyze the connectivity of the components in the circuit. Once the regularity has been identified, it can be used to simplify the analysis and design of the circuit, or to generate a more compact representation of the circuit for storage or transmission.

Regularity can be useful in the analysis and optimization of a circuit, as it can allow for the use of repetitive structures and simplify the design process. With that being clear, how is regularity extracted?

3.1 Existing Work on Regularity Extraction

To begin with, regularity extraction is a process key to achieving high quality, efficient and low-cost digital designs for a number of reasons. For example, by identifying and exploiting regularity in a design, the resulting implementation is more structured and predictable, which can improve the overall quality of the design. Another essential benefit of regular structures, achieved by grouping cells or modules together, is the area utilization. Improving the last, results in smaller area and lower power consumption.

Until today, several approaches have been developed on how to extract and measure regularity within the circuit. Some of them include graph traversals and matching approaches, while others include template-based approaches, as well as signature-based methods.

3.1.1 Graph Traversals and Matching Approaches

As far as graph traversals are concerned, in prior approaches, the key was to model the original circuit into a bipartite graph, in order to use a bipartite edge-cover algorithm to address the problem, as stated in [1]. Each bipartite graph represents a bit-slice path, as shown in fig. 3.1.

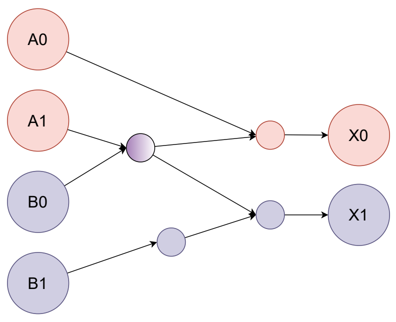


Figure 3.1: 4-to-2 Sliced Datapath [1]

On the other hand, several converts the bit slice problem into bit matching problem, like in [3]. The motivation of this method is based on the fact that although each datapath can be different, there may exist a main path between a starting point and its corresponding ending point. Once the starting points are able to match to ending points, the gates between a pair of matched points can get extracted to make one bit slice.

3.1.2 Template-based Approaches

The fundamental step of template-based approaches is the generation of a large set of templates, where a template is a sub-circuit with multiple instances in the circuit. In this section, two algorithms will be mentioned, one for templates with a tree structure and one for a special class of multi-output templates, called *single-principal-output* templates, where all outputs of a template are in the transitive fanin of a particular output [4].

Based on [4], regularity in a given circuit can be of three types, functional, structural or topological. For example, given a high level description, a functional-regular circuit uses a set of functionally-equivalent operations or sub-circuits. Functional regularity is an essential first step towards the generation of a compact and regular template. A structurally regular description can be represented schematically by assigning a horizontal or vertical direction to the nets. Finally, a topologically regular design consists of an ordered set of blocks, which gives a good initial placement for the circuit. For the algorithms that will be mentioned, functional regularity in high-level descriptions is concerned.

The regularity extraction problem is composed as a graph with input a circuit C of logic components that can be either small logic blocks, such as AND or OR gates, multiplexers or arithmetic blocks, such as adders and shifters. This circuit is represented by a directed graph, where the nodes correspond to the logic components or the primary inputs and the edges correspond to the interconnection between them. Having formulated the problem, it can now be stated as follows:

1. Regularity Extraction Problem [4]: *Given a circuit represented by a graph G (V,E), find a cover C(G)=$\{G_1, G_2, ..., G_n\}$, which is partitioned into m templates $S_1, S_2, ..., S_m$, such that the number n of sub-graphs and the overall area of the templates are maximized.*

2. Template Generation Problem [4]: *Given a circuit represented by a graph G(V,E), generate the complete set of templates where each template has at least two sub-graphs.*

Generation of Tree Templates

A *tree template* is a template with a single output and no internal reconvergence. First, the nodes of graph G are topologically sorted, and for every pair of nodes, a template is generated with two sub-graphs, one rooted at each node. After that, the logic functions of the two nodes are compared in order to construct the list of children templates. The template S_m is then compared to previously constructed ones, with a binary search, and if it is equivalent with others, its sub-graphs are added to the existing one. This process is repeated for every node-pair.

Generation of Multi-Output Templates

This algorithm is the extension of the algorithm for the tree templates. A disclaimer is that only those multi-output sub-graphs whose every output lies in the transitive fanin of a particular output are used, and they are called *single PO sub-graphs*. A single PO sub-graph is represented by a list of its nodes in depth-first-search, and that is because all isomorphic sub-graphs are unique. For node w and root node u, there can be multiple paths through different incoming edges of u, so, the first thing is to define a *list of path*, containing the indices of the incoming edges of u through which w is connected to u. After that, the nodes in node-list of G_u and G_v are compared pairwise. If the path-list of the corresponding nodes are different, they are removed from the sub-graphs and finally, if they are the same, the remaining copies are removed from the node-list.

3.1.3 Signature-based Methods

Signature-based regularity extraction approaches are methods for identifying patterns or regularities in data by using predefined "signatures" or templates. In this report, an algorithm that analyzes the circuit connectivity, to recognize regularity, and then automatically extracts regular structures is going to be mentioned. This algorithm [5] can handle two design types:

1. Designs with a portion of the datapath components identified at the HDL level.
    - Structured cluster information for these components is identified in the netlist.
2. Designs with no such structured cluster information.
    - Such information for datapath components is automatically created by using hints from bus names and datapath features.

Regularity Extraction Problem [5]**:** *Group random instances into regular functions such that the given objective function is optimized.*

The signature of a random instance is dictated by its master cell and its connectivity to datapath instances. For example, the random instance **A** of the figure 3.2 is connected to datapath instance **P** at terminal point **x** and to datapath **Q** at terminal point **y**. So, the signature for instance A will be:

$$S(A) = f(M(A), h(M(P), x), h(M(Q), y)),$$

where f and h are properly chosen hash functions and the function M maps cell instances to their master cells.

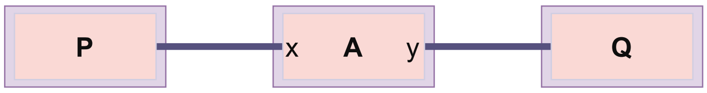


Figure 3.2: Signature definition

3.1.4 Structured Clustering Algorithms

In this report, **Structured Clustering Algorithms** [2] will be used. The goal of Structural Clustering algorithms is to create cell groups, based on netlist connectivity. But first, in order to explain the algorithms, some scientific terms need to be clear.

To begin with, **a cluster** is a set of connected cellgroups and **seed cellgroups** constitute the starting point of the process. In prior approaches (section 3.1.1), seed groups were generated by identifying registers with common control signals, while here, they are generated by Flip-Flop (FF) bit bus name. This can happen because FF instance names are preserved from RTL to netlist. After seed groups are generated, they are grown backwards into clusters, with the algorithms that will be discussed below, and formulate cell groups of the same size without the need to be of identical library cell type.

Cluster Representation & SDP Placement

To represent a cluster, a tree of size same as the cellgroups' size is used, with unique cells in each cellgroup. Thus, a cluster of size N, contains N isomorphic, single cell member trees, one per cell group member. In the figure below, there is a representation of a cluster tree with 20 tree nodes, thus 20 cellgroups.

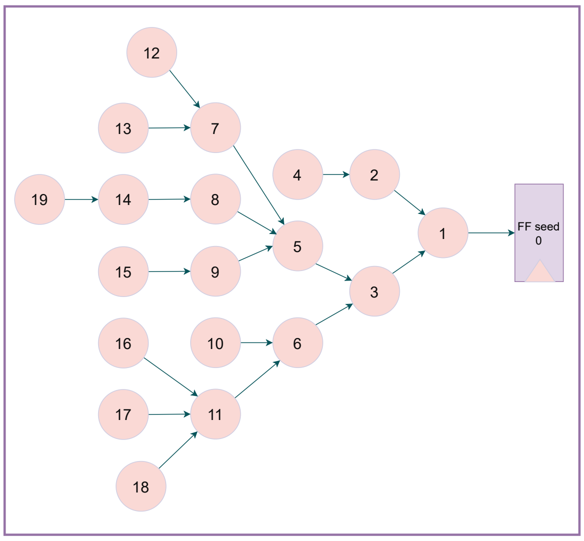


Figure 3.3: Tree of Cellgroups [2]

If the cellgroup size is 5, the resultant cluster size is $20 * 5 = 100$ cells and the cluster is composed by five sets of cluster trees, like the one in the figure above (figure 3.2).

As far as the cluster placement process is concerned, the idea of centre of mass combined with a set of cluster parameters is used. These parameters are:

1. **Row, Column (R, C) Organisation.** This can be either, i) multiple rows, single column, or ii) single row, multiple columns, or finally iii) a custom R, C organisation

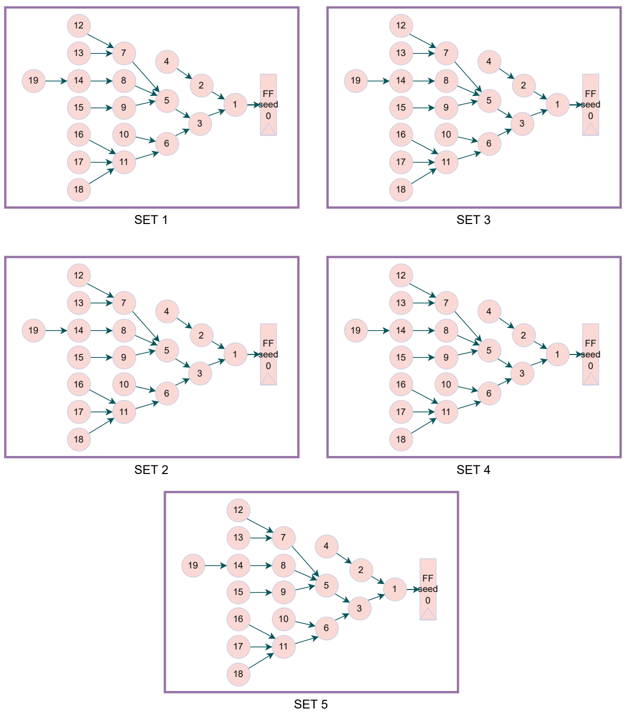


Figure 3.4: Resultant Cluster for group size 5 [2]

matching the core area aspect ratio.

2. **x, y strides.** Stride is called the required horizontal and vertical spacing between the members of a cellgroup.

3. **x,y origin.** Origin is defined as the x, y coordinates in the core area that the cellgroups is going to be placed.

Cluster tree cellgroups are placed one by one, in tree traversal order, at the nearest legal position. As mentioned above, for the cells that belong to a cellgroup, there is no need to be of the same library cell type, thus the size of each cell may differ. This situation generates a problem for the R, C organisation and especially for the columns. If the cellgroup size is 5, and the wanted organisation is multiple columns, single row, C would be 5. In order for this problem to be eliminated, this 5 (C) is multiplied by the size of the biggest cell, while if it was multiplied by anything rather than that, the resulted columns would not be enough to fit all cells. This example is illustrated in the figure below.

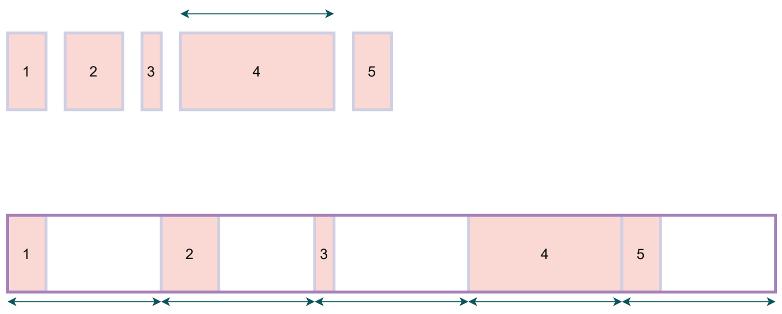


Figure 3.5: Cellgroup "Column size" explanation

This solution may result in empty space between cells. This space could be filled with unclustered cells later, in the placement process. For the example in fig. 3.3, the possible RC organizations could be:

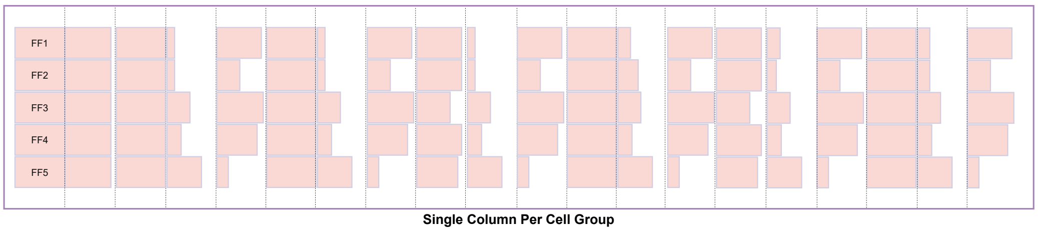


Figure 3.6: Single Column Per Cellgroup [2]

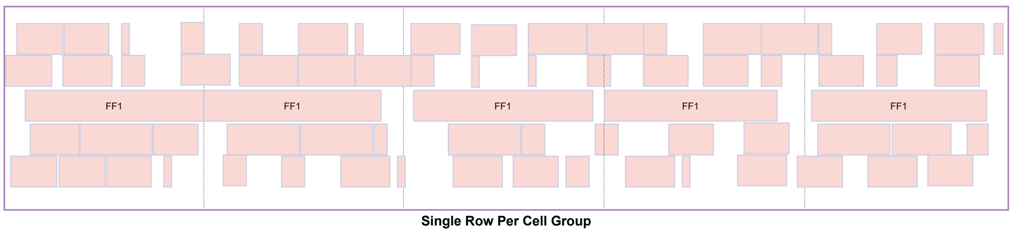


Figure 3.7: Single Row Per Cellgroup [2]

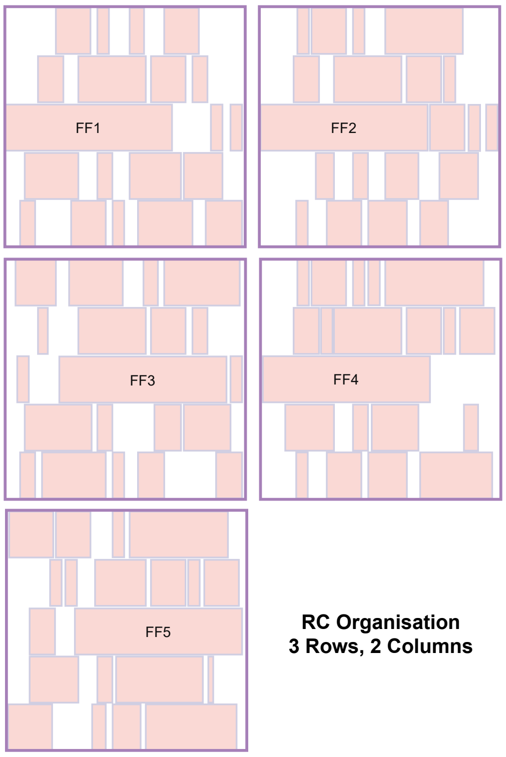


Figure 3.8: Multiple Row, Multiple Columns [2]

Now, let's see further the two extraction algorithms that will be used.

Greedy Clustering

During the greedy approach, every member's connected components are explored backwards in the circuit graph and attempt to find new members that will be added to the cluster tree representation. The basic principle, also mentioned above, is that one cell can only belong to one cellgroup. So, the reason that this algorithm is called *greedy* is that it makes local decisions about the common components. That is, buffers or gates shared between logic cones of FF seed groups are excluded from clusters, as clustered cells must be unique. However, if common components are excluded, the matching size is not optimal. Isomorphism based clustering algorithm is here to better solve this problem.

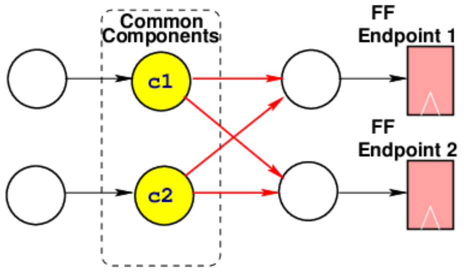


Figure 3.9: Common Components Across Endpoints [2]

Isomorphism Based Clustering

Due to graph isomorphism algorithm's NP complexity, in order to use it, it is transformed into a Forest of Trees Isomorphism problem. That is, the circuit graph is broken down into a forest of trees by routing new trees at fanout points of the original graph. These forests are then grown backwards, one per FF endpoint logic cone and the goal is to find the maximum isomorphism across the forests. The question for the common components assignment arises and the answer is that it is performed by solving the maximum bipartite matching problem, using Edmonds-Karp algorithm, on the formulated bipartite graph of common components and FF endpoints.

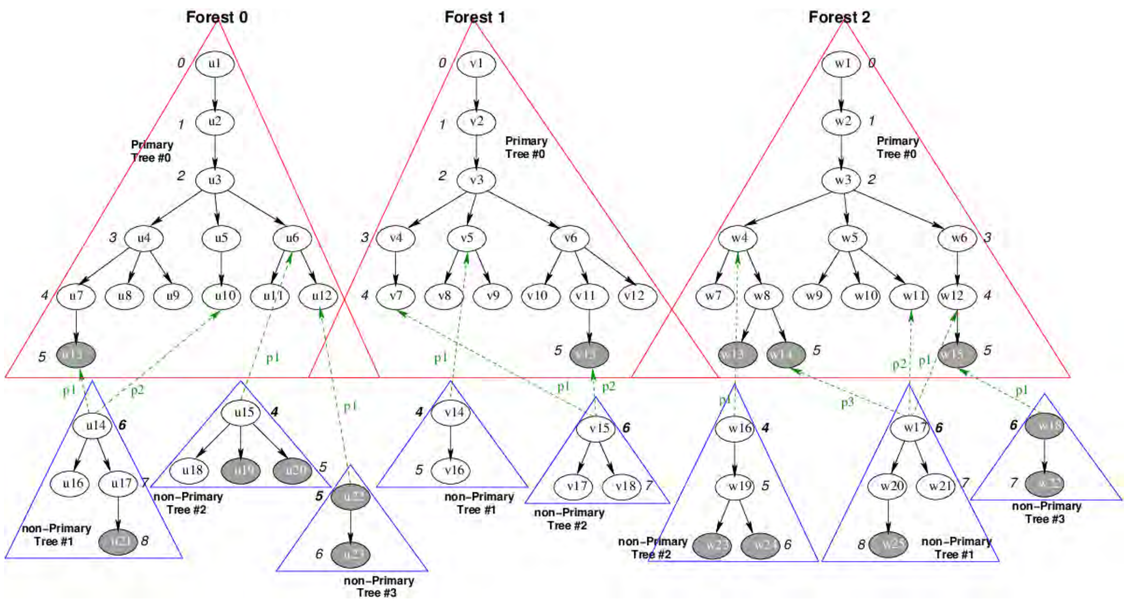


Figure 3.10: Forest of Trees Matching Problem Example [2]

Chapter 4

Verilog RTL Impact on Regularity

4.1 The Power of Designer's Verilog Knowledge

4.1.1 What is Verilog?

Verilog is a hardware description language (HDL) used to design digital circuits and systems. It is used to describe the behavior and structure of digital circuits and systems, allowing the designer to describe the functionality of the circuit in a manner that is similar to the way it would be described in a programming language.

Verilog can be used to design and simulate a wide range of digital circuits and systems, including processors, memory systems, communication systems, and digital signal processing systems. It is widely used in the electronic design automation (EDA) industry, and is supported by a wide range of EDA tools and simulators.

Verilog provides a set of constructs such as modules, always and initial blocks, and tasks and functions that allow the designer to describe the behavior of the system at various levels of abstraction, from the gate level to the algorithmic level. It also provides a rich set of data types, such as integers, logic vectors, and arrays, which can be used to describe the data that flows through the system.

4.1.2 Synthesis Process

From **RTL Verilog** all the way to **netlist**, intercedes a process called **synthesis**. It practically takes the high-level description of a digital circuit and converts it into a gate-level representation of the circuit. The synthesis process typically involves a number of steps. For starters, the RTL implementation is parsed and checked for syntax errors. After that, the design is expanded to include any instantiated modules or components and proceeds to technology mapping. There, the design is mapped to the specific target technology, such as the type of FPGA or ASIC being used. This step may include performing logic optimization to reduce the number of gates used in the final design. Placement and routing are following, in order to determine the physical location of the gates and the routing of the interconnections between them. The final output (the netlist) cannot be generated unless timing analysis is performed. During timing analysis, the design passes through checks to make sure it meets the timing constraints specified in the HDL code. The netlist can be used for programming of programmable devices (FPGA or CPLD) or for the manufacturing process of custom ASICs. It's worth noting that the synthesis process is performed by specialized software called synthesis tool, which may be provided by the device vendor or by third-party vendors.

During synthesis process, it is very important how RTL is written, for the netlist that will be generated. For example, a multiplexer (MUX) described with logic gates in RTL, it is going to have the same gates after synthesis in the netlist, while if it is implemented with if-else or case statement, the logic gates of the netlist will be different and totally up to synthesizer. To better understand this situation, there is an example below.

The verilog code for a 4 to 1, 1-bit multiplexer, using *a case* statement, can be seen in the listing below.

```
module MUX4to1_case (input a,          // 1-bit input
                     input b,          // 1-bit input
                     input c,          // 1-bit input
                     input d,          // 1-bit input
                     input [1:0] sel,  // input sel used to select
    between a,b,c,d
                     output out);      // 1-bit output based on input
    sel

  reg out;

  always @ (a or b or c or d or sel) begin
    case (sel)
      2'b00 : out <= a;
      2'b01 : out <= b;
      2'b10 : out <= c;
      2'b11 : out <= d;
    endcase
  end

endmodule
```

Listing 4.1: 4 to 1 MUX with case statement

After synthesis with GENUS Cadence Tool [6], with a virtual clock, the netlist has the two following library cells:

```
// Generated by Cadence Genus(TM) Synthesis Solution 18.10-p003_1
module MUX4to1_case(a, b, c, d, sel, out);
  input a, b, c, d, sel;
  output out;
  wire a, b, c, d, sel;
  wire out;
  wire n_0;
  inverter g34(.I (n_0), .ZN (out));
  multiplexer g35__8780(.I0 (a), .I1 (b), .S (sel), .ZN (n_0));
endmodule
```

Listing 4.2: Netlist of 4 to 1 MUX with case statement

Due to RTL's freedom of writing, synthesizer found that these two cells could implement the task successfully. On the other hand, for the verilog description of the MUX below, listing 4.3, the gates are specific. The synthesizer will have to look at the library for cells that could fulfill these gates' requirements.

```verilog
module MUX4to1_gates (input a,          // 1-bit input
                      input b,          // 1-bit input
                      input c,          // 1-bit input
                      input d,          // 1-bit input
                      input [1:0] sel,  // input sel used to select
    between a,b,c,d
                      output out);      // 1-bit output based on input
    sel

  reg out;
  wire s1n, s0n, y1, y2, y3;

  not G1( s1n, sel[1]); // first not gate
  not G2( s0n, sel[0]); // second not gate

  // instantiating and gates
  and  G3(y0, a, s1n, s0n);
  and  G4(y1, b, s1n, s0);
  and  G5(y2, c, s1, s0n);
  and  G6(y3, d, s1, s0);

  // instantiating or gate
  or G7(out, y0, y1, y2, y3);
endmodule
```

Listing 4.3: 4 to 1 MUX implemented with gates

After synthesis with GENUS Cadence Tool [6], with a virtual clock, the netlist has the three following library cells:

```verilog
// Generated by Cadence Genus(TM) Synthesis Solution 18.10-p003_1
module MUX4to1_gates(a, b, c, d, sel, out);
  input a, b, c, d;
  input [1:0] sel;
  output out;
  wire a, b, c, d;
  wire [1:0] sel;
  wire out;
  wire n_0, n_1;
  nor g25__8780(.A2 (sel[0]), .A1 (n_1), .ZN (out));
  nand g26__4296(.A2 (a), .A1 (n_0), .ZN (n_1));
  inverter g27(.I (sel[1]), .ZN (n_0));
endmodule
```

Listing 4.4: Netlist of 4 to 1 MUX with gates

After not finding cells with the specific function, the synthesizer searched for the closest ones. That is, these two RTL representations of a 4 to 1, 1-bit MUXs, implement the same design, though with different cells. For the specific example, there is no big difference, due to its size. Image how the area and the timing could be affected for a bigger design if the RTL description is not optimal.

With that being said, it is of great importance to write "good" verilog code. The term "good" in the context of regularity extraction is translated to *techniques to improve regularity.*

4.2 Techniques that Improve Netlist's Regularity

As mentioned in section 3.1.4, in this report, Structured Clustering Algorithms will be used and especially **Isomorphism Based Clustering**. The key to increase regularity is to understand **common components**, why they exist in the specific place and how to deal with them. Reminder, isomorphism algorithm grows cellgroups backwards until it finds common components.

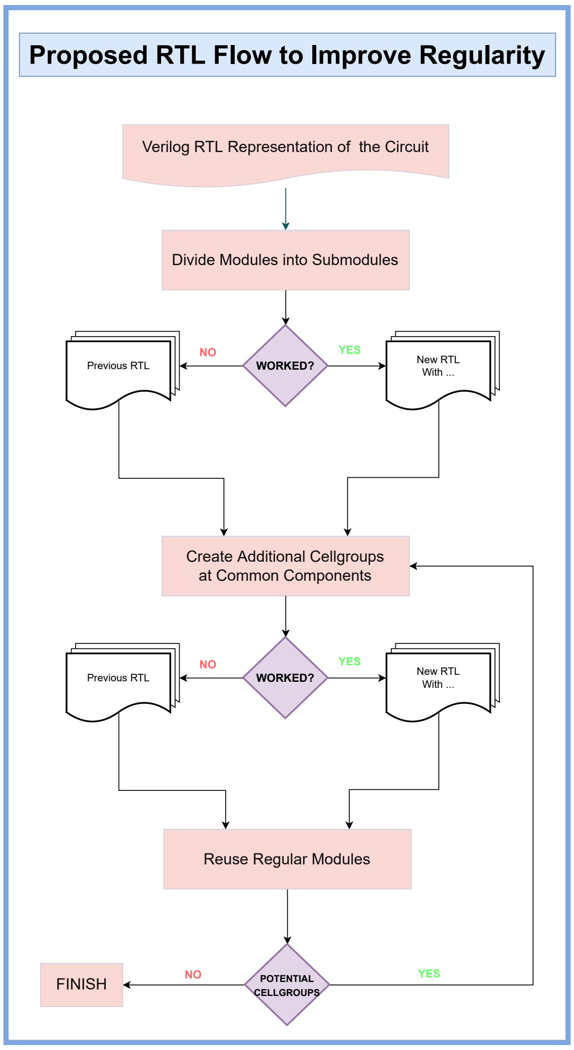

Proposed RTL Flow to Improve Regularity
Verilog RTL Representation of the Circuit
Divide Modules into Submodules
Previous RTL
NO
WORKED?
YES
New RTL With ...
Create Additional Cellgroups at Common Components
Previous RTL
NO
WORKED?
YES
New RTL With ...
Reuse Regular Modules
FINISH
NO
POTENTIAL CELLGROUPS
YES

Figure 4.1: RTL Techniques Flow

In the figure fig. 4.1, the **RTL changes proposed flow** is given. It begins with the RTL representation of the circuit and passes through the first step, *module separation* (section 4.2.1). After that, either this technique increased the regularity, or not, it passes through the second step, *cellgroup creation*(section 4.2.2). This step is very important because the algorithm grows back clusters only from seed groups, which contain registers. But what if the design does not contain registers at all or there are a lot of common components and growing back, stops immediately? In this case, it is necessary to have understood the behavioral RTL implementation in order to be able after synthesis, in the netlist, to create lists of cells, that is **cellgroups**.

Disclaimer: *This step does not exactly belong in the verilog RTL based techniques, but due to the fact that it is fundamental to have understood the behavioral implementation, I inserted it in this flow step.*

Similarly, whether the second step increased the regularity or not, flow proceeds to the third step, *reuse of already regular modules* (section 4.2.3). At the end of that step, it is highly possible that there is room for addition cellgroups creation, thus it returns to the second step and repeats the process. If there is no room for additional optimizations to increase regularity, verilog is ready to be synthesized. The desired percentage of regularity is, approximately, above 70%.

In order to make the techniques that are going to be proposed, more obvious, an example will be shown, for each one of the methods. These examples are part of the main case study of this report. Succinctly, the case studies, further analyzed later, are:

- Register files
  - flip-flops ($\times$**32** *AND* $\times$**64**)
- **Booth Multiplier**
  - Synthetic adder ($\times$**2**)
  - MUX Tree ($\times$**16**)
  - **CSA tree (CSA core *OR* CSA$\times$16)**
- DLX execute unit

Carry Save Adder, member of the CSA tree of booth multiplier, will help analyze the first technique.

4.2.1 First Flow Step - Module Separation

Carry Save Adder (CSA)

A carry-save adder (CSA) is a type of digital circuit that is used to add multiple numbers together. It is called a "carry-save" adder because it generates two outputs: a "sum" output and a "carry" output. The sum output represents the sum of the input numbers without the carry-out bits, and the carry output represents the carry-out bits from the addition of the input numbers.

The basic building block of a CSA is a full adder, which is a circuit that can add three inputs: A, B, and Cin (carry-in) and produces two outputs: S (sum) and Cout (carry-out). The full adder circuit performs the following operation:

$$S = A \oplus B \oplus Cin \tag{4.1}$$

$$Cout = (A\&B)|(Cin\&(A \oplus B)) \tag{4.2}$$

A CSA can be constructed by cascading multiple full adder circuits together, with the sum output of each full adder being connected to the carry-in input of the next full adder. The number of full adder circuits used in a CSA will depend on the number of bits in the input numbers. For example, a 4-bit CSA would require 4 full adder circuits.

The advantage of a CSA over a conventional adder is that it can perform the addition of multiple numbers in parallel, which can significantly increase the speed of the addition operation.

RTL Implementation of original CSA & Regularity Measurements

The implementation of a CSA is very simple and consists of two basic functions, xor and majority (equations 4.1 and 4.2). So, the verilog code for a 64 bit carry save adder will be the listing 4.5:

```
module originalCSA(s1, s2, p1, p2, p3, clk);
  input clk;
  input [63:0] p1,p2,p3;
  output reg [63:0] s1,s2;

  always @(posedge clk)
  begin
    s1 = p1 ^ p2 ^ p3;
    s2 = (p1 & p2) | (p2 & p3) | (p3 & p1);
  end
endmodule
```

Listing 4.5: Original CSA

Disclaimer: *The only reason registers are used, instead of just wires is to be easier for the structured clustering isomorphism algorithm, to extract the cellgroups, based on the flip-flop name.* After synthesizing the above circuit for 1 nanosecond clock period at a 250 nanometer library and loading the resulting netlist at ASP tool [7], the extracted regularity is:

```
> INFO: Total Cells = 192, Total Grouped Cells = 128, Grouped Cell Ratio = 67%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 25288.70um^2,
> Total Grouped Cells Area = 17160.19um^2, Grouped Cell Area Ratio = 68%
```

As shown, Grouped Cell Ratio is equal to 67%. This percentage is not bad, mainly due to the fact that the design is really small. The reason it stops growing back the cluster is, as mentioned before, the common components. In fact, the resulted cellgroup is only one, containing the registers of S_1 and S_2, that is, the **seed group**. In listing 4.6 there is the cellgroups report extracted from ASP tool.

```
 ################################################################
# Generated by: PathVIZ (Version: 1.0)
# OS: Linux x86_64(Host: torreyridge)
# Generated on: Sun Jan 29 18:40:19 2023
# Design: originalCSA
################################################################
```

```
# Core Utilisation: 70%
# Core Width, Height: 0.000, 0.000, Aspect Ratio: -nan
# Core X, Y Offsets: 0.000, 0.000
###############################################################
create_cellgroup -module originalCSA -groupid 1 -groupname originalCSA
    /\\s2_reg\[53\] {originalCSA/\\s2_reg\[53\] originalCSA/\\s1_reg
    \[62\] originalCSA/\\s1_reg\[60\] originalCSA/\\s1_reg\[56\]
    originalCSA/\\s1_reg\[1\] originalCSA/\\s1_reg\[48\] originalCSA/\\
    s1_reg\[32\] originalCSA/\\s1_reg\[0\] originalCSA/\\s1_reg\[33\]
    originalCSA/\\s2_reg\[0\] originalCSA/\\s2_reg\[63\] originalCSA/\\
    s1_reg\[31\] originalCSA/\\s2_reg\[7\] originalCSA/\\s2_reg\[62\]
    originalCSA/\\s2_reg\[61\] originalCSA/\\s1_reg\[47\] originalCSA/\\
    s1_reg\[30\] originalCSA/\\s2_reg\[60\] originalCSA/\\s2_reg\[59\]
    originalCSA/\\s1_reg\[29\] originalCSA/\\s2_reg\[11\] originalCSA/\\
    s2_reg\[58\] originalCSA/\\s2_reg\[57\] originalCSA/\\s1_reg\[55\]
    originalCSA/\\s2_reg\[13\] originalCSA/\\s1_reg\[46\] originalCSA/\\
    s1_reg\[28\] originalCSA/\\s2_reg\[56\] originalCSA/\\s2_reg\[16\]
    originalCSA/\\s2_reg\[55\] originalCSA/\\s1_reg\[27\] originalCSA/\\
    s2_reg\[54\] originalCSA/\\s2_reg\[50\] originalCSA/\\s1_reg\[45\]
    originalCSA/\\s1_reg\[26\] originalCSA/\\s2_reg\[52\] originalCSA/\\
    s2_reg\[19\] originalCSA/\\s2_reg\[51\] originalCSA/\\s1_reg\[25\]
    originalCSA/\\s1_reg\[63\] originalCSA/\\s2_reg\[21\] originalCSA/\\
    s2_reg\[49\] originalCSA/\\s1_reg\[59\] originalCSA/\\s1_reg\[54\]
    originalCSA/\\s2_reg\[24\] originalCSA/\\s1_reg\[44\] originalCSA/\\
    s1_reg\[24\] originalCSA/\\s2_reg\[48\] originalCSA/\\s2_reg\[47\]
    originalCSA/\\s1_reg\[23\] originalCSA/\\s2_reg\[46\] originalCSA/\\
    s2_reg\[45\] originalCSA/\\s2_reg\[28\] originalCSA/\\s1_reg\[43\]
    originalCSA/\\s1_reg\[22\] originalCSA/\\s2_reg\[44\] originalCSA/\\
    s2_reg\[30\] originalCSA/\\s2_reg\[43\] originalCSA/\\s1_reg\[21\]
    originalCSA/\\s2_reg\[42\] originalCSA/\\s1_reg\[16\] originalCSA/\\
    s2_reg\[41\] originalCSA/\\s1_reg\[53\] originalCSA/\\s1_reg\[42\]
    originalCSA/\\s2_reg\[22\] originalCSA/\\s2_reg\[40\] originalCSA/\\
    s2_reg\[39\] originalCSA/\\s1_reg\[19\] originalCSA/\\s2_reg\[38\]
    originalCSA/\\s2_reg\[37\] originalCSA/\\s1_reg\[41\] originalCSA/\\
    s1_reg\[18\] originalCSA/\\s2_reg\[36\] originalCSA/\\s2_reg\[35\]
    originalCSA/\\s1_reg\[17\] originalCSA/\\s2_reg\[34\] originalCSA/\\
    s2_reg\[17\] originalCSA/\\s1_reg\[61\] originalCSA/\\s1_reg\[58\]
    originalCSA/\\s1_reg\[52\] originalCSA/\\s1_reg\[40\] originalCSA/\\
    s2_reg\[32\] originalCSA/\\s2_reg\[31\] originalCSA/\\s1_reg\[15\]
    originalCSA/\\s2_reg\[29\] originalCSA/\\s1_reg\[39\] originalCSA/\\
    s1_reg\[14\] originalCSA/\\s2_reg\[27\] originalCSA/\\s1_reg\[13\]
    originalCSA/\\s2_reg\[26\] originalCSA/\\s2_reg\[25\] originalCSA/\\
    s1_reg\[51\] originalCSA/\\s1_reg\[38\] originalCSA/\\s1_reg\[12\]
    originalCSA/\\s2_reg\[23\] originalCSA/\\s1_reg\[11\] originalCSA/\\
    s1_reg\[20\] originalCSA/\\s1_reg\[37\] originalCSA/\\s1_reg\[10\]
    originalCSA/\\s2_reg\[20\] originalCSA/\\s1_reg\[9\] originalCSA/\\
    s2_reg\[18\] originalCSA/\\s2_reg\[33\] originalCSA/\\s1_reg\[57\]
    originalCSA/\\s1_reg\[50\] originalCSA/\\s1_reg\[36\] originalCSA/\\
    s1_reg\[8\] originalCSA/\\s2_reg\[15\] originalCSA/\\s1_reg\[7\]
    originalCSA/\\s2_reg\[14\] originalCSA/\\s1_reg\[35\] originalCSA/\\
    s1_reg\[6\] originalCSA/\\s2_reg\[12\] originalCSA/\\s1_reg\[5\]
    originalCSA/\\s2_reg\[10\] originalCSA/\\s2_reg\[9\] originalCSA/\\
    s1_reg\[49\] originalCSA/\\s1_reg\[34\] originalCSA/\\s1_reg\[4\]
    originalCSA/\\s2_reg\[8\] originalCSA/\\s1_reg\[3\] originalCSA/\\
    s2_reg\[6\] originalCSA/\\s2_reg\[5\] originalCSA/\\s1_reg\[2\]
    originalCSA/\\s2_reg\[4\] originalCSA/\\s2_reg\[3\] originalCSA/\\
    s2_reg\[2\] originalCSA/\\s2_reg\[1\]}
# -flatsortedgroupid 0

# initialise group placement constraints after new groups are
    instantiated
 remove_groups_placement_constraints

 # verify that all specified cellgroup components are indeed unique
 check_cellgroups
```

Listing 4.6: Original CSA's cellgroups

The common components here exist because the synthesizer uses the optimal cells from the library based on the RTL given. As a result, and mainly to optimize the circuit, it uses logic that merges S_1 and S_2 and common gates between them, that they stop the cluster from going back. So, the proposal here to prevent this from happening is:

Divide modules into submodules, separate them when it is necessary, in order to force the synthesizer to withhold the hierarchy and not optimize similar logic as common parts of the same logic cone.

Quote 1: First Proposed Technique - Module's Separation

RTL Implementation of modified CSA & Regularity Measurements

In order to test the *Quote 1*, modified CSA is implemented, with the two main functions (equations 4.1 and 4.2) in separate modules. So, the verilog code for the 64 bit modified carry save adder will be:

```verilog
module splittedCSA(s1, s2, p1, p2, p3, clk);
  input clk;
  input [63:0] p1,p2,p3;
  output reg [63:0] s1, s2;

  wire [63:0] s1_wire, s2_wire;

  XOR xor_inst (.p1(p1),
                .p2(p2),
                .p3(p3),
                .s1(s1_wire));

  MAJORITY majority_inst (.p1(p1),
                          .p2(p2),
                          .p3(p3),
                          .s1(s2_wire));

  always @(posedge clk )
  begin
    s1 <= s1_wire;
    s2 <= s2_wire;
  end
endmodule
```

Listing 4.7: Modified CSA

where *XOR* and *MAJORITY* the modules in listings listing 4.8 and listing 4.9 respectively.

```verilog
module XOR(p1, p2, p3, s1);
  input [63:0] p1, p2, p3;
  output [63:0] s1;

  assign s1 = p1 ^ p2 ^ p3;
endmodule
```

Listing 4.8: XOR Submodule

```verilog
module MAJORITY(p1, p2, p3, s1);
  input [63:0] p1, p2, p3;
  output [63:0] s1;

  assign s1 = (p1 & p2) | (p2 & p3) | (p3 & p1);
endmodule
```

Listing 4.9: MAJORITY Submodule

Again, *the only reason registers are used is to be easier for the structured clustering isomorphism algorithm, to extract the cellgroups, based on the flip-flop name.*

After synthesizing the above circuit for 1 nanosecond clock period at a 250 nanometer library and loading the resulting netlist at ASP tool [7], the extracted regularity is:

```
> INFO: Total Cells = 386, Total Grouped Cells = 384, Grouped Cell Ratio = 99%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 27998.21um^2,
> Total Grouped Cells Area = 27998.21um^2, Grouped Cell Area Ratio = 100%
```

As shown, Grouped Cell Ratio is equal to 99%. By forcing the synthesizer to withhold the hierarchy, it is not trying to optimize it, using gates that will satisfy both S_1 and S_2 function. This results in better regularity measurements.

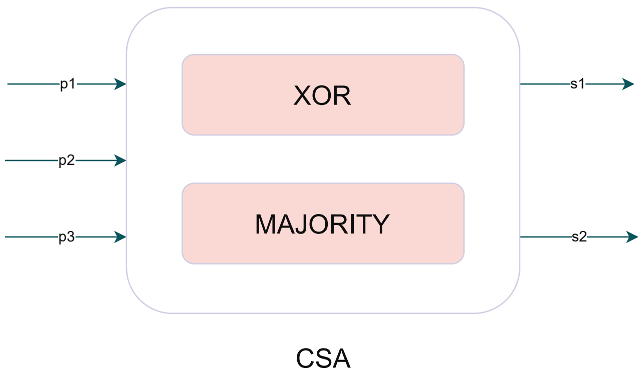


Figure 4.2: Schematic of CSA with Separated Modules

In fig. 4.2, there is the CSA instantiation with XOR and MAJORITY submodules. Both XOR and MAJORITY take the same inputs, p_1, p_2 and p_3 and state as output s_1

and s_2 respectively. Growing back from them, thanks to the submodules, there will be no common components amongst them. This is the reason why regularity reaches 99%.

4.2.2 Second Flow Step - Additional Cellgroups Creation

Continuing with the flow, the second step is the creation of additional cellgroups, regarding the netlist synthesized from step one (section 4.2.1). For the specific example, carry save adder, the regularity is already 99%, so let me extend it to the CSA Core, a tree of sixteen CSAs. For this design, (section 4.2.2), as the flow indicates, module separation comes first, and it is followed by cellgroup creation. Despite the result, it proceeds to step three, reuse of regular modules and back to step 2 to check if there is room for additional cellgroup creation. For convenience, and in order to maintain the flow of this report and to not go back and forth, to explain this step, the netlist after step three is going to be used, that is, the netlist with the already maximum usage of regular modules. How this process is executed and what the results are, is going to be shown later, in section 4.2.3. But firstly, **CSA Core** will be explained.

Booth Multiplier Core - CSA Tree

CSA tree consists of sixteen CSA adders, connected in levels. As CSA is an adder 3-to-2, meaning that it corresponds 3 inputs to 2 outputs, the number of levels of sixteen such adders will be 5. The verilog implementation is given in the listing 4.10.

```
module CSA_core_splitted(sum0, sum1, sum2, sum3, sum4, sum5, sum6, sum7,
     sum8, sum9, sum10, sum11, sum12, sum13, sum14, sum15, s15_ff, c15_ff
    , clk, reset);

input clk, reset;

// CSA inputs //
input [63:0] sum0, sum1, sum2, sum3, sum4, sum5, sum6, sum7, sum8, sum9,
     sum10, sum11, sum12, sum13, sum14, sum15;

// CSA outputs //
wire [63:0] s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13,
    s14, s15;
wire [63:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13,
    c15, c15;

output reg [63:0] s15_ff;
output reg [63:0] c15_ff;

// first CSA level //
splittedCSA csa_0 (.s1(s0),.s2(c0),.p1(sum0),.p2(sum1),.p3(sum2));
splittedCSA csa_1 (.s1(s1),.s2(c1),.p1(sum3) p2(sum4),.p3(sum5));
splittedCSA csa_2 (.s1(s2),.s2(c2),.p1(sum6),.p2(sum7),.p3(64'b0));
splittedCSA csa_3 (.s1(s3),.s2(c3),.p1(sum8),.p2(sum9),.p3(sum10));
splittedCSA csa_4 (.s1(s4),.s2(c4),.p1(sum11),.p2(sum12),.p3(sum13));
splittedCSA csa_5 (.s1(s5),.s2(c5),.p1(sum14),.p2(sum15),.p3(64'b0));

// second CSA level //
splittedCSA csa_6 (.s1(s6),.s2(c6),.p1(s0),.p2({c0[62:0],1'b0}),.p3(s1))
    ;
splittedCSA csa_7 (.s1(s7),.s2(c7),.p1({c1[62:0], 1'b0}),.p2(s2),.p3({c2
    [62:0], 1'b0}));
```

```
splittedCSA csa_8 (.s1(s8),.s2(c8),.p1(s3),.p2({c3[62:0], 1'b0}),.p3(s4)
    );
splittedCSA csa_9 (.s1(s9),.s2(c9),.p1({c4[62:0], 1'b0}),.p2(s5),.p3({c5
    [62:0], 1'b0}));

// third CSA level //
splittedCSA csa_10 (.s1(s10),.s2(c10),.p1(s6),.p2({c6[62:0], 1'b0}),.p3(
    s7));
splittedCSA csa_11 (.s1(s11),.s2(c11),.p1(s8),.p2({c8[62:0], 1'b0}),.p3(
    s9));

// fourth CSA level //
splittedCSA csa_12 (.s1(s12),.s2(c12),.p1(s10),.p2({c10[62:0], 1'b0}),.
    p3({c7[62:0], 1'b0}));
splittedCSA csa_13 (.s1(s13),.s2(c13),.p1(s11),.p2({c11[62:0], 1'b0}),.
    p3({c9[62:0], 1'b0}));

// fifth CSA level //
splittedCSA csa_14 (.s1(s14),.s2(c14),.p1(s12),.p2({c12[62:0], 1'b0}),.
    p3(s13));

// last CSA level //
splittedCSA csa_15 (.s1(s15),.s2(c15),.p1(s14),.p2({c14[62:0], 1'b0}),.
    p3({c13[62:0], 1'b0}));

always @(posedge clk or posedge reset)
begin
  if (reset == 1'b1)
  begin
    s15_ff <= 64'd0;
    c15_ff <= 64'd0;
  end
  else
  begin
    s15_ff <= s15;
    c15_ff <= c15;
  end
end

endmodule
```

Listing 4.10: CSA Tree

The CSA tree schematic is in the fig. 4.3. For the internal CSAs, separated CSA is used. That is, from the registers s15_ff and c15_ff, it will grow back to CSA_15 and due to XOR and MAJORITY submodules, as shown in section 4.2.1, the algorithm will not find any common components until it reaches the inputs of CSA_15. There, due to the fact that both submodules have the same inputs, the growing will stop. This happens because, seed groups, the start points of the growing, are by default, groups with registers, thus all 64 bits of s15_ff and c15_ff (listing 4.11). That is, although the substructure is ready, there are no cellgroups to start from and grow back. The resulting regularity is shown in block 4.2.2.

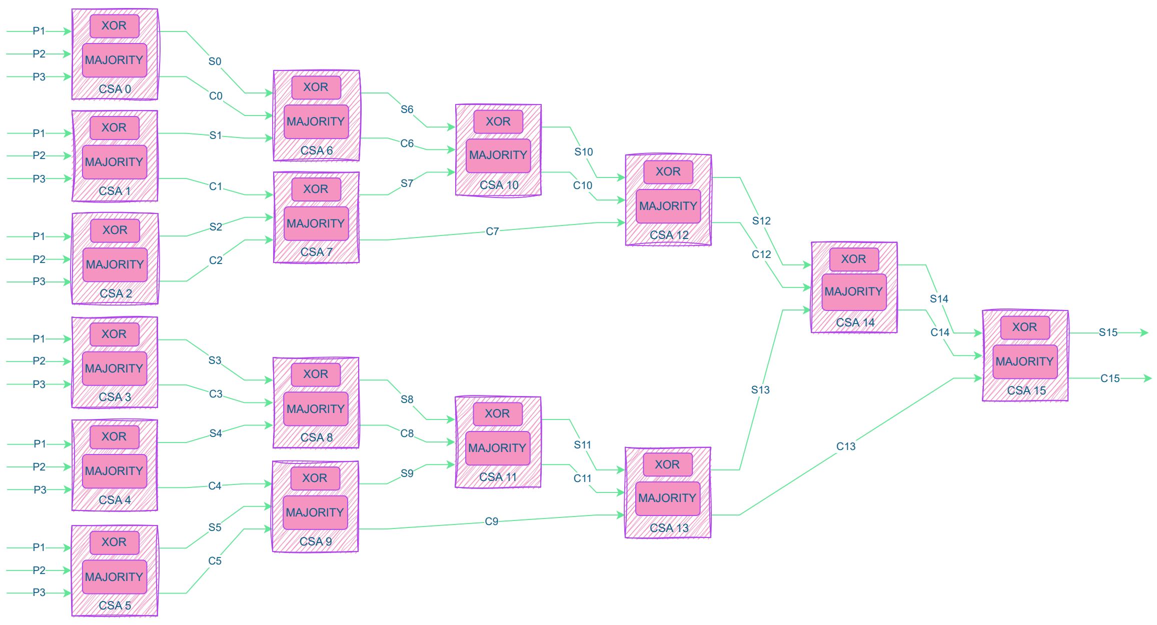


Figure 4.3: CSA Tree with CSAs from section 4.2.1

```
create_cellgroup -module CSA_core_splitted -groupid 1 -groupname
    CSA_core_splitted/\\c15_ff_reg\[17\] {CSA_core_splitted/\\c15_ff_reg
    \[17\] CSA_core_splitted/\\c15_ff_reg\[63\] CSA_core_splitted/\\
    c15_ff_reg\[62\] CSA_core_splitted/\\c15_ff_reg\[61\]
    CSA_core_splitted/\\c15_ff_reg\[60\] CSA_core_splitted/\\c15_ff_reg
    \[59\] ................ CSA_core_splitted/\\c15_ff_reg\[5\]
    CSA_core_splitted/\\c15_ff_reg\[4\] CSA_core_splitted/\\c15_ff_reg
    \[3\] CSA_core_splitted/\\c15_ff_reg\[2\]}
# -flatsortedgroupid 1

create_cellgroup -module CSA_core_splitted -groupid 2 -groupname
    CSA_core_splitted/\\s15_ff_reg\[62\] {CSA_core_splitted/\\s15_ff_reg
    \[62\] CSA_core_splitted/\\s15_ff_reg\[60\] CSA_core_splitted/\\
    s15_ff_reg\[56\] CSA_core_splitted/\\s15_ff_reg\[48\]
    CSA_core_splitted/\\s15_ff_reg\[32\] CSA_core_splitted/\\s15_ff_reg
    \[31\] ................ CSA_core_splitted/\\s15_ff_reg\[2\]
    CSA_core_splitted/\\s15_ff_reg\[1\] CSA_core_splitted/\\s15_ff_reg
    \[61\] CSA_core_splitted/\\s15_ff_reg\[0\]}
# -flatsortedgroupid 0
```

Listing 4.11: CSA Tree Seed Groups

```
> INFO: Total Cells = 4222, Total Grouped Cells = 633, Grouped Cell Ratio = 15%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 190504.94um^2,
> Total Grouped Cells Area = 39421.87um^2, Grouped Cell Area Ratio = 21%
```

The solution here, is to create the cellgroups that will play the role of seed-groups, to continue growing back. So, for every CSA, two cellgroups have to be created, one to grow the XOR sub-module (listing 4.12) back and one for the MAJORITY(listing 4.13).

```
create_cellgroup -module XOR -groupid 1 -groupname XOR_0 {XOR/g12490 XOR
   /g12492 XOR/g2 XOR/g12914 XOR/g12919 XOR/g12921 XOR/g12922 XOR/g12929
    XOR/g12930 XOR/g12933 XOR/g12934 XOR/g12935 XOR/g12936 XOR/g12937
   XOR/g12938 XOR/g12939 XOR/g12940 XOR/g12941 XOR/g12947 XOR/g12950 XOR
   /g12951 XOR/g12953 XOR/g12954 XOR/g12955 XOR/g12958 XOR/g12961 XOR/
   g12967 XOR/g12974 XOR/g12978 XOR/g12980 XOR/g12981 XOR/g12983 XOR/
   g12986 XOR/g12988 XOR/g12989 XOR/g12990 XOR/g12994 XOR/g13000 XOR/
   g13003 XOR/g13005 XOR/g13006 XOR/g13007 XOR/g13008 XOR/g13009 XOR/
   g13013 XOR/g13015 XOR/g13016 XOR/g13017 XOR/g13018 XOR/g13020 XOR/
   g13021 XOR/g13022 XOR/g13023 XOR/g13025 XOR/g13027 XOR/g13029 XOR/
   g13030 XOR/g13031 XOR/g13032 XOR/g13033 XOR/g13034 XOR/g13035 XOR/
   g13036 XOR/g13038}
```

Listing 4.12: XOR Seed Group

```
create_cellgroup -module MAJORITY -groupid 1 -groupname MAJORITY_0 {
   MAJORITY/g9470 MAJORITY/g9471 MAJORITY/g9472 MAJORITY/g9473 MAJORITY/
   g9474 MAJORITY/g9475 MAJORITY/g9476 MAJORITY/g9477 MAJORITY/g9478
   MAJORITY/g9479 MAJORITY/g9480 MAJORITY/g9481 MAJORITY/g9482 MAJORITY/
   g9483 MAJORITY/g9484 MAJORITY/g9485 MAJORITY/g9486 MAJORITY/g9487
   MAJORITY/g9488 MAJORITY/g9489 MAJORITY/g9490 MAJORITY/g9491 MAJORITY/
   g9492 MAJORITY/g9493 MAJORITY/g9494 MAJORITY/g9495 MAJORITY/g9496
   MAJORITY/g9497 MAJORITY/g9498 MAJORITY/g9499 MAJORITY/g9500 MAJORITY/
   g9532 MAJORITY/g9533 MAJORITY/g9534 MAJORITY/g9535 MAJORITY/g9536
   MAJORITY/g9537 MAJORITY/g9538 MAJORITY/g9539 MAJORITY/g9540 MAJORITY/
   g9541 MAJORITY/g9542 MAJORITY/g9543 MAJORITY/g9544 MAJORITY/g9545
   MAJORITY/g9546 MAJORITY/g9547 MAJORITY/g9548 MAJORITY/g9549 MAJORITY/
   g9550 MAJORITY/g9551 MAJORITY/g9552 MAJORITY/g9553 MAJORITY/g9554
   MAJORITY/g9555 MAJORITY/g9556 MAJORITY/g9557 MAJORITY/g9558 MAJORITY/
   g9559 MAJORITY/g9560 MAJORITY/g9561 MAJORITY/g9562 MAJORITY/g9563}
```

Listing 4.13: MAJORITY Seed Group

These two cellgroups contain the cells that are connecting to the output of each sub-module. After creating all wanted cellgroups and growing back the clusters, the resulted regularity is:

```
> INFO: Total Cells = 4222, Total Grouped Cells = 4171, Grouped Cell Ratio = 99%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 190504.94um^2,
> Total Grouped Cells Area = 190427.33um^2, Grouped Cell Area Ratio = 100%
```

As shown, Grouped Cell Ratio is equal to 99%. By creating cellgroups where common components are found, gives the algorithm more space to grow back and significantly increase the regularity. This leads to the second conclusion:

Create cellgroups, where common components are found, in order to behave like seed-groups.

Quote 2: Second Proposed Technique - Cellgroup Creation

4.2.3 Third Flow Step - Reuse of Regular Modules

Last but not least, it is very important to reuse already regular modules for regularity increment. If a circuit has multiple instantiations of the same logic function, it is preferred to try to regularize one of them and then reuse it.

CSA Tree

In order to clarify this proposition, once again, CSA tree example will be used. CSA tree, used in modified radix-4 32x32 booth multiplier, as explained above, consists of sixteen carry save adders, connected in levels. The verilog RTL implementation, using the original CSA, is in the listing 4.14, where "originalCSA" the listing 4.5.

```
module CSA_core_original(sum0, sum1, sum2, sum3, sum4, sum5, sum6, sum7,
    sum8, sum9, sum10, sum11, sum12, sum13, sum14, sum15, s15_ff, c15_ff
    , clk, reset);

input clk, reset;

// CSA inputs //
input [63:0] sum0, sum1, sum2, sum3, sum4, sum5, sum6, sum7, sum8, sum9,
    sum10, sum11, sum12, sum13, sum14, sum15;

// CSA outputs //
wire [63:0] s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13,
    s14, s15;
wire [63:0] c0, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13,
    c15, c15;

output reg [63:0] s15_ff;
output reg [63:0] c15_ff;

// first CSA level //
originalCSA csa_0 (.s1(s0),.s2(c0),.p1(sum0),.p2(sum1),.p3(sum2));
originalCSA csa_1 (.s1(s1),.s2(c1),.p1(sum3) p2(sum4),.p3(sum5));
originalCSA csa_2 (.s1(s2),.s2(c2),.p1(sum6),.p2(sum7),.p3(64'b0));
originalCSA csa_3 (.s1(s3),.s2(c3),.p1(sum8),.p2(sum9),.p3(sum10));
originalCSA csa_4 (.s1(s4),.s2(c4),.p1(sum11),.p2(sum12),.p3(sum13));
originalCSA csa_5 (.s1(s5),.s2(c5),.p1(sum14),.p2(sum15),.p3(64'b0));

// second CSA level //
originalCSA csa_6 (.s1(s6),.s2(c6),.p1(s0),.p2({c0[62:0],1'b0}),.p3(s1))
    ;
originalCSA csa_7 (.s1(s7),.s2(c7),.p1({c1[62:0], 1'b0}),.p2(s2),.p3({c2
    [62:0], 1'b0}));
originalCSA csa_8 (.s1(s8),.s2(c8),.p1(s3),.p2({c3[62:0], 1'b0}),.p3(s4)
    );
originalCSA csa_9 (.s1(s9),.s2(c9),.p1({c4[62:0], 1'b0}),.p2(s5),.p3({c5
    [62:0], 1'b0}));

// third CSA level //
originalCSA csa_10 (.s1(s10),.s2(c10),.p1(s6),.p2({c6[62:0], 1'b0}),.p3(
    s7));
originalCSA csa_11 (.s1(s11),.s2(c11),.p1(s8),.p2({c8[62:0], 1'b0}),.p3(
    s9));

// fourth CSA level //
```

```
originalCSA csa_12 (.s1(s12),.s2(c12),.p1(s10),.p2({c10[62:0], 1'b0}),.
    p3({c7[62:0], 1'b0}));
originalCSA csa_13 (.s1(s13),.s2(c13),.p1(s11),.p2({c11[62:0], 1'b0}),.
    p3({c9[62:0], 1'b0}));

// fifth CSA level //
originalCSA csa_14 (.s1(s14),.s2(c14),.p1(s12),.p2({c12[62:0], 1'b0}),.
    p3(s13));

// last CSA level //
originalCSA csa_15 (.s1(s15),.s2(c15),.p1(s14),.p2({c14[62:0], 1'b0}),.
    p3({c13[62:0], 1'b0}));

always @(posedge clk or posedge reset)
begin
  if (reset == 1'b1)
  begin
    s15_ff <= 64'd0;
    c15_ff <= 64'd0;
  end
  else
  begin
    s15_ff <= s15;
    c15_ff <= c15;
  end
end
```

Listing 4.14: Tree of Original Carry Save Adders

The CSA tree schematic is in the fig. 4.4.

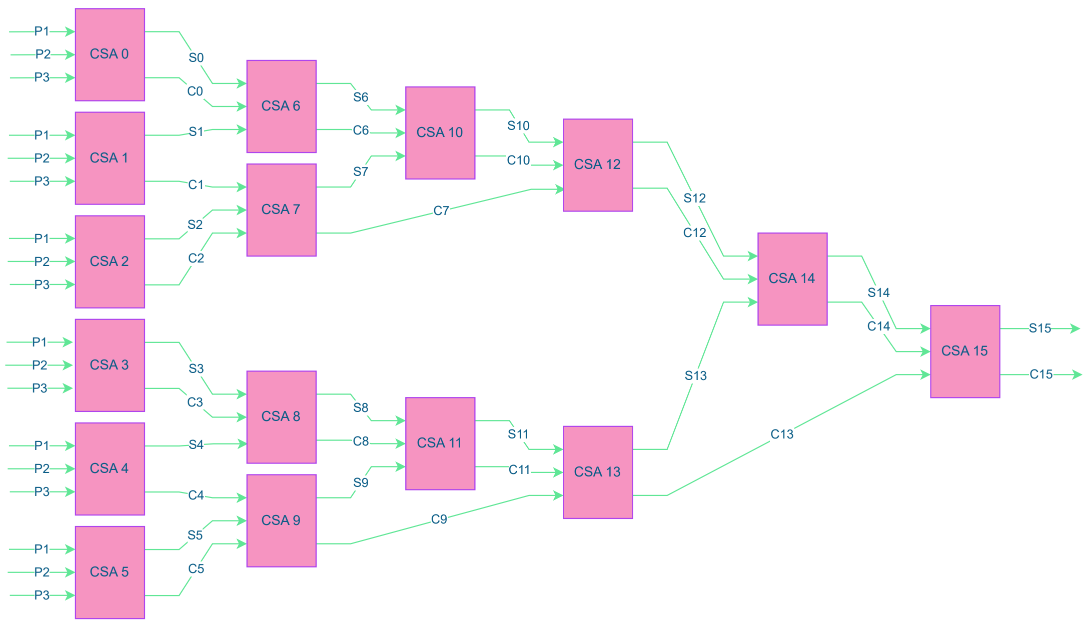


Figure 4.4: CSA Tree with Original CSAs

The regularity measurements for the above design, synthesized at 4 nanoseconds, is:

```
> INFO: Total Cells = 1583, Total Grouped Cells = 190, Grouped Cell Ratio = 12%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 143921.23um^2,
> Total Grouped Cells Area = 27603.07um^2, Grouped Cell Area Ratio = 19%
```

In section 4.2.1 the CSA was regularized by dividing adder module to two submodules, xor and majority. So, in this section, original CSA is going to be replaced by separated CSA module in the process of reusing regular modules. For starters, only the last one of all sixteen adders is going to be replaced by the regular one. After synthesizing at 4 nanoseconds period and loading the netlist at ASP tool [7], the regularity measurement is:

```
> INFO: Total Cells = 1771, Total Grouped Cells = 381, Grouped Cell Ratio = 22%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 146179.15um^2,
> Total Grouped Cells Area = 29903.33um^2, Grouped Cell Area Ratio = 20%
```

Comparatively, to the previous analysis, regularity has increased 10% by replacing only one of the sixteen adders. Now, all adders are going to be replaced like in the schematic below:

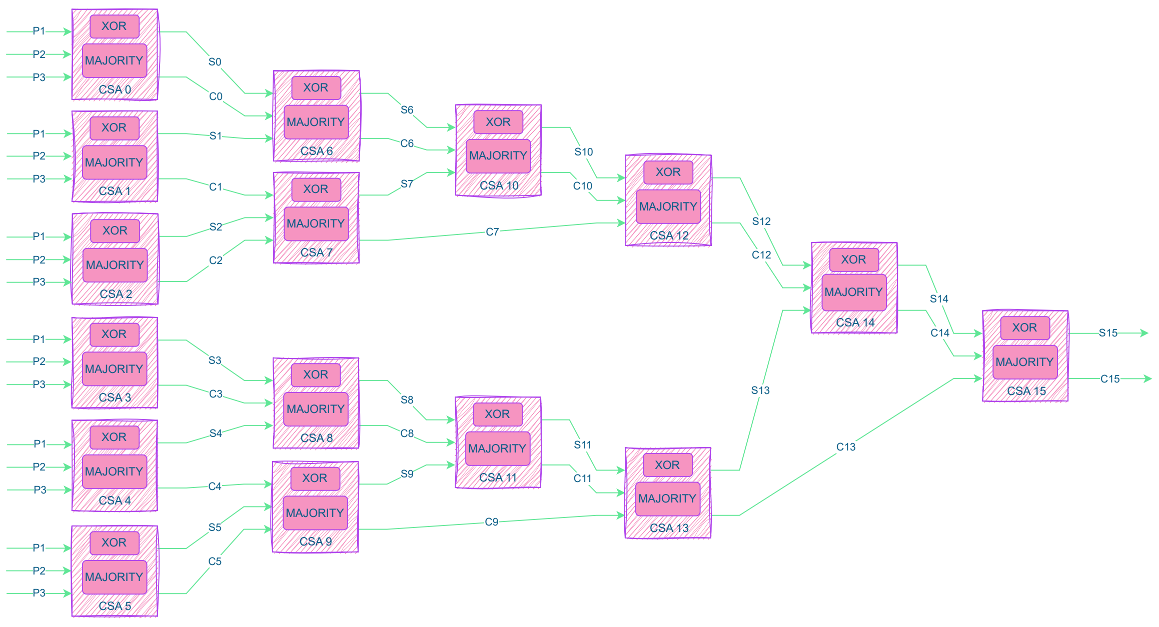


Figure 4.5: Schematic of CSA Tree with Splitted Modules

For the specific design, although all adders are replaced by the regular ones, regularity will not increase more. This happens because now it is necessary to create the additional cellgroups, discussed in section 4.2.2. The point of the process is:

> **When instantiations of the same module appear in the design more than one time, try and regularize a single one and reuse it for the rest of them.**

Quote 3: Third Proposed Technique - Reuse of Regular Modules

DLX Execute Unit

The execute unit is a significant part of the DLX processor architecture, but first, let me say a few words about DLX itself.

DLX Processor is a reference model of a Reduced Instruction Set Computing (RISC) microprocessor. The DLX, like the MIPS design, bases its performance on the use of an instruction pipeline. This pipeline, is a sequence of stages through which instructions are executed, and is designed to increase the performance of the processor by allowing multiple instructions to be processed in parallel. A typical DLX processor pipeline consists of the following stages:

- Instruction Fetch (IF): In this stage, the instruction to be executed is fetched from memory and stored in an instruction register.
- Instruction Decode (ID): In this stage, the instruction is decoded and the operands are read from the register file.
- Execution (EX): In this stage, the ALU performs the arithmetic or logical operation specified by the instruction.
- Memory Access (MEM): In this stage, the processor accesses memory to read or write data.
- Write Back (WB): In this stage, the result of the operation is written back to the register file.

Each stage of the pipeline takes one clock cycle to complete, so an instruction can be processed in multiple stages in parallel. However, the pipeline also introduces the potential for stalls and hazards, which can reduce performance if not properly handled.

Back to the **Execution Unit**(EU), also known as the Arithmetic Logic Unit (ALU), is a key component of the DLX processor. Its primary function is to perform the arithmetic and logical operations specified by the instructions. The EU is responsible for performing operations such as addition, subtraction, bitwise operations (e.g. AND, OR, NOT), and other functions required by the instruction set architecture (ISA) of the DLX processor.

The EU is typically composed of several functional units, including a set of arithmetic logic circuits and a set of register files. The arithmetic logic circuits perform the actual arithmetic and logical operations, while the register files store the intermediate results. The EU also has inputs for the operands, which are either read from the register file or fetched from memory, and outputs for the results, which are written back to the register file or memory.

DLX instructions can be broken down into three types, R-type, I-type and J-type. R-type instructions are register-register instructions that perform arithmetic and logical operations on data stored in the register file, while I-type instructions are register-immediate instructions that perform arithmetic and logical operations on data stored in the register file and an immediate value. Finally, J-type instructions are jump instructions that change the program counter to a target address. Each type of instruction has a unique opcode that identifies the instruction and specifies its operation. The DLX ISA defines the syntax and semantics of each instruction, including the number and type of operands and the operation performed by the instruction. The instruction formats are used to efficiently encode and decode instructions in the DLX processor.

In all of the three types, an adder design is widely used for the functions that are performed. For example, for the I-type:

```
case(IR_opcode_field)
    `LW   :  ALU_result = reg_out_A + Imm;
    `LH   :  ALU_result = reg_out_A + Imm;
    `LB   :  ALU_result = reg_out_A + Imm;
    `LBU  :  ALU_result = reg_out_A + Imm;
    `LHU  :  ALU_result = reg_out_A + Imm;
    ...
```

Listing 4.15: Example of I-type instructions

After synthesizing DLX EU for three nanosecond clock period and loading into ASP tool [7], the regularity measurements are below:

```
> INFO: Total Cells = 2958, Total Grouped Cells = 359, Grouped Cell Ratio = 12%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 101811.02um^2,
> Total Grouped Cells Area = 19855.58um^2, Grouped Cell Area Ratio = 20%
```

Now, suppose there is an adder design, regular enough, that could replace those adders (+) in the EU. In total, there are ten adders (+). By replacing them with the regular one, the RTL implementation will look like:

```
module EXcustomADDERS (ALU_result, reg_out_B_EX, reg_dst_out,
   mem_write_EX, mem_read_EX, mem_to_reg_EX, reg_write_EX, opcode_of_EX,
    opcode_of_EX_reg, reg_dst_of_EX, clk, reset, IR_opcode_field,
   IR_function_field, reg_out_A, reg_out_B, Imm, rt_addr, rd_addr,
   reg_dst, reg_write, mem_to_reg, mem_read, mem_write, byte, word,
   counter, noop);
...
// custom adders instantiation //
simple_adder_wire #(.k(32)) LW (.a(reg_out_A), .b(Imm), .sum(lw));
simple_adder_wire #(.k(32)) LH (.a(reg_out_A), .b(Imm), .sum(lh));
simple_adder_wire #(.k(32)) LB (.a(reg_out_A), .b(Imm), .sum(lb));
simple_adder_wire #(.k(32)) LBU (.a(reg_out_A), .b(Imm), .sum(lbu));
simple_adder_wire #(.k(32)) LHU (.a(reg_out_A), .b(Imm), .sum(lhu));
...

always @(posedge clk)
begin
...
    case(IR_opcode_field)
```

```
        `LW   :  ALU_result = lw;
        `LH   :  ALU_result = lh;
        `LB   :  ALU_result = lb;
        `LBU  :  ALU_result = lbu;
        `LHU  :  ALU_result = lhu;
        ...
    endcase
    ...
end
...
endmodule
```

Listing 4.16: Part of EU reusing regular adder

After synthesizing DLX EU for four nanosecond clock period and loading into ASP tool [7], the regularity measurements are below:

```
> INFO: Total Cells = 7316, Total Grouped Cells = 2714, Grouped Cell Ratio = 37%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 233391.31um^2,
> Total Grouped Cells Area = 92595.89um^2, Grouped Cell Area Ratio = 40%
```

Grouped Cell Ratio increased from 12% to 37%, without modifying the RTL, by just reusing an already regular module. This result indicates the significance of the third flow step. But how did this adder become regular? What is underneath the *simple_adder_wire* module used in listing 4.16? These questions will be answered in the next chapter.

Chapter 5

Regularity Extraction & Measurements using Logic Cones

5.1 What happens in case of an Agnostic Designer?

In the previous chapter, it was made pretty clear that regularity depends a lot on the verilog RTL design, that is, the designer's knowledge and the use of verilog. But what happens if the designer is agnostic? What happens if the designer wants on purpose to let the synthesis tool find the best implementation of a specific design of a circuit?

Usually, this depends on the requirements and constraints of the design. If there are specific requirements for the circuit, such as performance, power consumption, or specific component constraints, then writing a specific design can help meet those requirements. However, if the purpose is to maximize performance and find the most efficient implementation, using a synthesizer to find the best implementation can be beneficial. Ultimately, the choice between a specific design and using a synthesizer will depend on the specific needs of the project and the goals of the designer.

That is, either the designer is agnostic or wants to let the synthesis tool meet the requirements and constraints of the design, there must be developed a way of extracting regularity for general designs that their verilog could not be changed. The term *general designs* stands for behavioral RTL. For example, there are multiple ways to implement an adder and various different architectures, like:

- **Ripple Carry Adder**: This is the simplest form of an adder, in which the carry generated in one stage is propagated to the next stage.
- **Carry Lookahead Adder**: This type of adder uses a carry-lookahead logic to generate carry bits in parallel, which reduces the delay in carry propagation and increases the speed of the adder.
- **Bounded Delay Adder**: This adder architecture is designed to provide a guaranteed maximum delay for the carry propagation.
- **Kogge-Stone Adder**: This is a high-speed parallel prefix adder that uses a pipelined architecture to reduce the critical path delay.
- **Brent-Kung Adder**: This is a high-speed parallel prefix adder that uses a recursive structure to reduce the number of gates required.

- **Wallace Tree Adder**: This is a high-speed pipelined adder that uses a combination of carry-lookahead and carry-skip techniques to reduce the delay in carry propagation.

And of course there is the behavioral implementation as well. Like mentioned above, the behavioral implementation of an adder is just the symbol +. The "+" symbol is a general mathematical symbol for addition and is used to perform arithmetic addition in various contexts. If "+" symbol is used without specifying a particular adder architecture, the resulting design, will depend on the context in which it is used. For example, in digital electronics, the "+" symbol may represent a simple ripple carry adder, a carry-lookahead adder, or any other type of adder architecture. In this case, no RTL changes can be done to increase the regularity, because both sum and carry chains are hidden and depend entirely on the synthesizer.

For that reason, a flow has been developed for cases, like the adder above, that the bit dependencies are hidden, to try and extract regularity. It may be design-specific, but it is also general enough for the specific design.

5.2 Case Studies

5.2.1 33-bit Synthesized Adder

An adder is a digital circuit that performs arithmetic operations, specifically addition. It typically consists of two chains: the sum chain and the carry chain. The sum chain generates the least significant bit (LSB) of the sum, while the carry chain generates the carry-out signal that is used to compute the next higher-order bit of the sum.

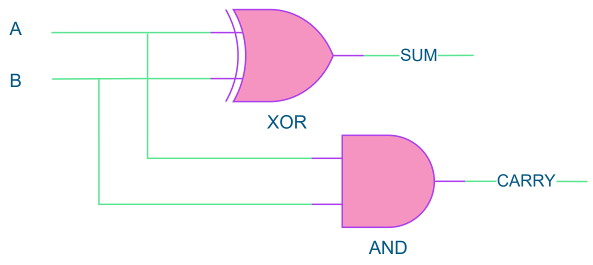


Figure 5.1: Half Adder

In fig. 5.1, there is a half-adder. A half adder is a simple type of binary adder that performs the addition of two single binary digits (0 or 1). It has two inputs, A and B, and two outputs, the sum and the carry. The sum is the exclusive OR (XOR) of the inputs, while the carry is the AND of the inputs. A half adder is called "half" because it only computes the least significant bit (LSB) of the sum, and not the carry bit that is used to compute the next higher-order bit. The half adder is a building block for creating larger adders, such as full adders, which can handle the addition of multiple binary digits.

In the fig. 5.2, there is, respectively, a full adder. It has three inputs: A, B and Carry-in (Cin), and two outputs: Sum (S) and Carry-out (Cout). The Sum is the result of the modulo-2 addition of the inputs, and the Carry-out is the result of the carry from the addition. A full adder is essentially a combination of two half adders, where one computes the sum and the other computes the carry.

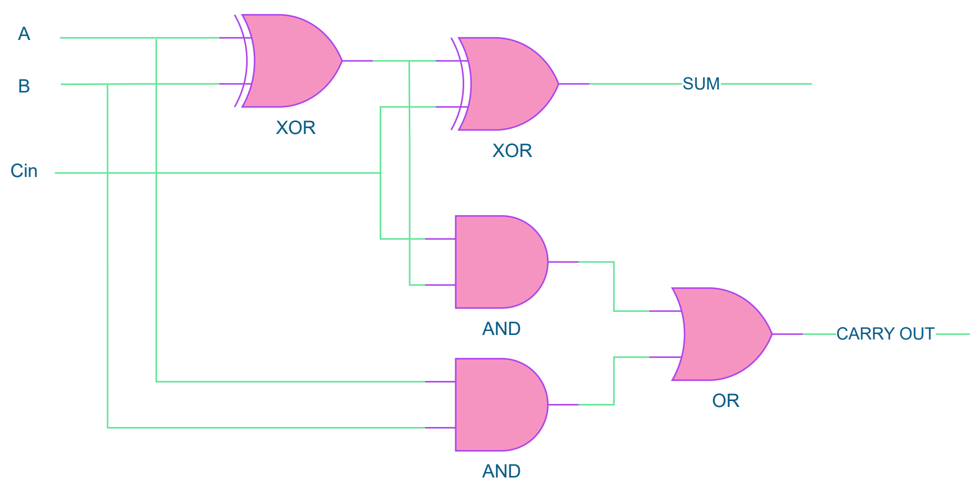


Figure 5.2: Full Adder

In the above two figures (fig. 5.1 and fig. 5.2), the sum and the carry chain, are pretty obvious.

Now, let's take a look at the listing 5.1.

```
module simple_adder_wire(a, b, sum);
  input [32:0] a, b;
  output [32:0] sum;

  assign sum = a + b;

endmodule
```

Listing 5.1: 33-bit Behavioural Adder

$$sum = a + b$$

The *sum* chain could be retrieved because of the output pins, while the *carry* chain is completely hidden. The only items that are known are the 33 bits of the two inputs, a and b and the 33 bits of the output, sum. Finding the backward logic cone from each bit of output sum, could possibly result in the sum chain. At the same time, the nth bit of sum will most definitely contain the nth bit of a and b. However, in this previous logic cone, there will also be parts of the carry chain, which, depending on the architecture, may be completely different for each sum bit. So, the need arises to separate the sum chain from the carry one. Up to this, the flow has taken the shape below:

1. For bit n, find the forward logic cone of input a (FLC_{an})
2. For bit n, find the forward logic cone of input b (FLC_{bn})
3. For bit n, find the backward logic cone of output sum (BLC_{sumn})
4. Concatenate list of FLC_{an} with list of FLC_{bn}

The next and final step to eliminate the carry bits from the sum chain is to **intersect** the two lists from items 3 and 4. That is,

5. Intersect list of (FLC_{an}, FLC_{bn}) with BLC_{sumn}

Each sum_n list, containing the bits on the nth sum chain, will be a cell-group.

Every step of the flow has been developed through TCL [8] functions and scripts that are being sourced in ASP tool [7].

```
set BITS 33

for {set n 0} {$n < $BITS} {incr n} {
    // step 1 //
    set a($n) [get_forward_logic_cone simple_adder_reg/a|$n -longest]
    // step 2 //
    set b($n) [get_forward_logic_cone simple_adder_reg/b|$n -longest]

    // step 4 //
    set ab($n) [concat $a($n) $b($n)]

    // step 3 //
    set sumBackward($n) [get_backward_logic_cone simple_adder_reg/\\
   sum_reg\[$n\]/D -longest]

    // step 5 //
    set SUM($n) [intersection $ab($n) $sumBackward($n)]

    set cellgroupSUM($n) [extract_pins_from_conesList $SUM($n)]

    create_cellgroup -module simple_adder_wire -groupid $n -groupname
   sum$n $cellgroupSUM($n)
}
```

Listing 5.2: Sum Extraction Flow

Commands used in steps 1, 2 and 3, *get_forward_logic_cone <pin_name> -longest* and *get_backward_logic_cone <pin_name> -longest* already exist in the ASP tool, and each one returns a list with the pins of the forward logic cone of *pin_name* and the backward one, respectively. Similarly, in step 4, command *concat*, exists already through the TCL interface. However, command *intersection*, used in step 5 does not pre-exist. The intersection of two lists in Tcl is the set of elements that are present in both lists. In other words, it's the common elements between the two lists. Intersecting two lists, may be complicated and its time complexity depends on the implementation used. One approach is to use the lsort command to sort both lists and then use two pointers to iterate through the sorted lists and compare elements. If a match is found, the element is added to the result list. Another approach is to use a loop to iterate through one list and check for elements in the other list using the lsearch command. This approach has a time complexity of O(n^2) in the worst case, where n is the length of the longer list.

The *cache* command is used to cache the results of a command, so that subsequent calls to the same command with the same arguments can be served from the cache, rather than executing the command again. If the lists being intersected are large and the operation is performed frequently, caching the result of the intersection operation with *cache* could help speed up the process by reducing the number of times the intersection operation needs to be performed.

So, combining a for loop to traverse the two lists and the *cache* command, the tcl procedure for the intersection is in the listing 5.3:

```
proc intersection {list1 list2} {
    set result {}

    foreach x $list1 {
        set cache($x) 1
    }

    foreach y $list2 {
        if {[info exists cache($y)]} {
            lappend result $y
        }
    }

    return $result
}
```

Listing 5.3: TCL List Intersection

Finally, at line 18 of listing 5.7, there is the last procedure needed for sum extraction. Procedure *extract_pins_from_conesList* is needed here, because commands get_forward_logic_cone and get_backward_logic_cone return lists with pins and not only the cells that will compose the cellgroup.

```
proc extract_pins_from_conesList {list} {
  set result {}
  set i 0

  foreach x $list {
    if {$i == 0} {
      incr i
    } else {
      if {[string last | $x] == -1} {
          // filter last appearence of '/' to locate the pin //
          set lastAT [string last / $x]
          set n [expr $lastAT-1]
          set final [string range $x 0 $n]

          // check if cell already exists, else append it //
          if {[lsearch -exact $result $final] < 0} {
            lappend result $final
          }
       }
    }
  }
  return $result
}
```

Listing 5.4: TCL Procedure for Pins Removal

The procedure in the listing 5.4, practically traverses the list given as argument, removes the pin from the cell and finally check if it already exists in the list. The possibility of already being on the list lies to the fact a cell has inputs and outputs. If a cell exists in the logic cone, both the input and its output will be in the list, resulting in duplicate entries.

So, after synthesizing listing 5.1 and loading into ASP tool [7], the regularity measurements are:

```
> INFO: Total Cells = 365, Total Grouped Cells = 192, Grouped Cell Ratio = 53%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 11063.81um^2,
> Total Grouped Cells Area = 6046.99um^2, Grouped Cell Area Ratio = 55%
```

Comparing to the regularity measurements without the *sum extraction process*, but only a cellgroup to play the role of seed group, containing the cells connected to the 33 bits of output sum:

```
> INFO: Total Cells = 365, Total Grouped Cells = 66, Grouped Cell Ratio = 18%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 11063.81um^2,
> Total Grouped Cells Area = 3062.30um^2, Grouped Cell Area Ratio = 28%
```

The cellgroup is in the listing 5.5.

```
create_cellgroup -module simple_adder_wire_DW01_add_1 -groupid 1 -
    groupname sum { simple_adder_wire_DW01_add_1/U376
    simple_adder_wire_DW01_add_1/U394 simple_adder_wire_DW01_add_1/U395
    simple_adder_wire_DW01_add_1/U397 simple_adder_wire_DW01_add_1/U399
    simple_adder_wire_DW01_add_1/U400 simple_adder_wire_DW01_add_1/U402
    simple_adder_wire_DW01_add_1/U405 simple_adder_wire_DW01_add_1/U406
    simple_adder_wire_DW01_add_1/U407 simple_adder_wire_DW01_add_1/U409
    simple_adder_wire_DW01_add_1/U415 simple_adder_wire_DW01_add_1/U427
    simple_adder_wire_DW01_add_1/U428 simple_adder_wire_DW01_add_1/U429
    simple_adder_wire_DW01_add_1/U432 simple_adder_wire_DW01_add_1/U448
    simple_adder_wire_DW01_add_1/U460 simple_adder_wire_DW01_add_1/U465
    simple_adder_wire_DW01_add_1/U475 simple_adder_wire_DW01_add_1/U476
    simple_adder_wire_DW01_add_1/U485 simple_adder_wire_DW01_add_1/U487
    simple_adder_wire_DW01_add_1/U492 simple_adder_wire_DW01_add_1/U498
    simple_adder_wire_DW01_add_1/U514 simple_adder_wire_DW01_add_1/U521
    simple_adder_wire_DW01_add_1/U522 simple_adder_wire_DW01_add_1/U527
    simple_adder_wire_DW01_add_1/U584 simple_adder_wire_DW01_add_1/U585
    simple_adder_wire_DW01_add_1/U586 simple_adder_wire_DW01_add_1/U587}
```

Listing 5.5: Seed Group for the 33-bit non-regular adder

On the other hand, finding the carry chain through logic cones is less complex. With n starting from 0, because C_{in} is zero,

1. For bit n, find the backward logic cone of output sum (BLC_{sumn})
2. For bit (n+1), find the backward logic cone of output sum ($BLC_{sum(n+1)}$)

3. For $carry_(n+1)$ Intersect list of $BLC_{sum(n+1)}$ with BLC_{sumn}

Each $carry_(n+1)$ list, containing the bits on the (n+1)th carry chain, will be a cellgroup.

Every step of the flow has been developed through TCL [8] functions and scripts that are being sourced in ASP tool [7].

```
1 set BITS 32
2
3 for {set n 0} {$n < $BITS} {incr n} {
4     // step 3, using results from step 3 of sum extraction flow //
5     set CARRY([expr $n + 1]) [intersection $sumBackward($n) $sumBackward([expr $n + 1])]
6     set cellgroupCARRY([expr $n + 1]) [extract_pins_from_conesList $CARRY([expr $n + 1])]
7 }
```

Listing 5.6: Carry Extraction Flow

The command used in step 3 is the *intersection* command from listing 5.3, and the command at line 6 is the command from listing 5.4.

However, depending on the adder architecture, each carry chain could contain cells from other carry chains. For example, for the 33 bit adder, carry chains 4 and 5 are:

```
CARRY(4): 8
CARRY(4): simple_adder_wire_DW01_add_1/U436 simple_adder_wire_DW01_add_1/U440 simple_adder_wire_DW01_add_1/U390 simple_adder_wire_DW01_add_1/U441 simple_adder_wire_DW01_add_1/U374 simple_adder_wire_DW01_add_1/U365 simple_adder_wire_DW01_add_1/U375 simple_adder_wire_DW01_add_1/U437
CARRY(5): 15
CARRY(5): simple_adder_wire_DW01_add_1/U557 simple_adder_wire_DW01_add_1/U439 simple_adder_wire_DW01_add_1/U412 simple_adder_wire_DW01_add_1/U356 simple_adder_wire_DW01_add_1/U357 simple_adder_wire_DW01_add_1/U436 simple_adder_wire_DW01_add_1/U361 simple_adder_wire_DW01_add_1/U413 simple_adder_wire_DW01_add_1/U440 simple_adder_wire_DW01_add_1/U390 simple_adder_wire_DW01_add_1/U441 simple_adder_wire_DW01_add_1/U374 simple_adder_wire_DW01_add_1/U365 simple_adder_wire_DW01_add_1/U375 simple_adder_wire_DW01_add_1/U437
```

Listing 5.7: Sum Extraction Flow

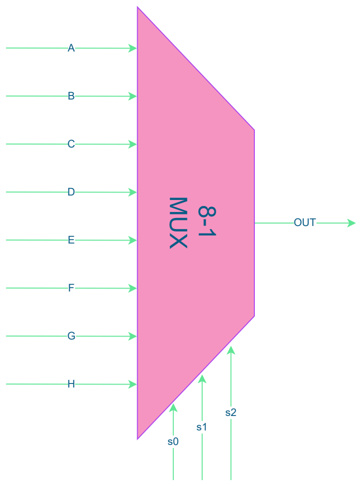
Carry chain 5 consists of every cell from carry chain 4 and seven more cells. That is, these lists are not unique, and in order to create cellgroups, lists must contain unique cells. The sum up, this process, the carry extraction flow, is not used to identify carry chains, make lists of them and subsequently, create cellgroups of them. This process is used to simply identify the carry chains, how their size increases along with the sum chains, in order to help later with the placement of the cellgroups.

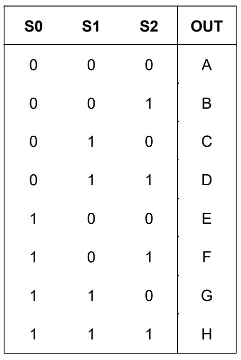
5.2.2 8-to-1 Synthesized Multiplexer

A multiplexer, also known as a MUX, is a device or circuit that selects one of several input signals and forwards the selected input to a single output line. The selection of the input is controlled by a set of selection lines. The design of a multiplexer involves specifying the number of inputs, the number of selection lines, the implementation of the logic that determines the input selection, and the choice of the technology used to implement the circuit.

A multiplexer can be implemented using various digital circuit elements such as gates, flip-flops, and decoders. The design of a multiplexer can also be described using a truth table, which shows the relationship between the inputs and outputs, or by using Boolean equations. In modern digital systems, multiplexers are often integrated into larger integrated circuits, and they are commonly used in communication systems, computer memory systems, and digital signal processing applications.

In fig. 5.3, there is an 8 to 1 multiplexer, the study of this subsection, and its truth table.





S0	S1	S2	OUT
0	0	0	A
0	0	1	B
0	1	0	C
0	1	1	D
1	0	0	E
1	0	1	F
1	1	0	G
1	1	1	H

Figure 5.3: 8 to 1 Multiplexer

In fig. 5.4, there is the diagram of a 1 bit 8 to 1 multiplexer, using gates.

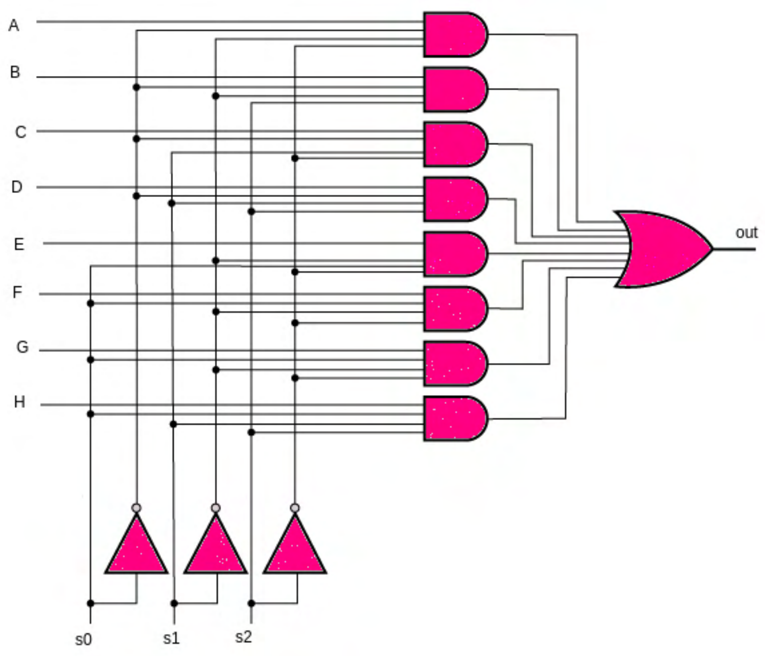


Figure 5.4: Logic Diagram of 1bit 8 to 1 Multiplexer

In this case study, the flow applied to the 33bit adder in section 5.2.1 will not work. This happens because a multiplexer design does not have a logic depth like the adder. Depending on the type of the MUX, 4 to 1, 8 to 1 etc, and the characteristics of the available gates of the library, like the number of inputs, the logic depth may differ, but its overall orientation is vertical, like shown in fig. 5.4. That is, grouping output "out" may not be the best approach.

The backward logic cone of output "out" for an 8-to-1 multiplexer (MUX) consists of all the inputs and control signals that affect the value of the output "out". In other words, it includes all the inputs and control signals that are necessary to determine the value of the output "out". For example:

- **Data inputs:** There would be 8 data inputs to the MUX, representing the 8 different data signals that can be selected as the output.
- **Selection inputs:** The MUX would have 3 selection inputs, which determine which of the 8 data inputs is selected as the output.
- **Output enable (OE) signal:** The MUX may have an output enable (OE) signal, which controls whether the MUX outputs the selected data signal or a default value (such as a high impedance state).

So, in summary, the backward logic cone of output "out" for an 8-to-1 MUX would typically consist of the 8 data inputs, the 3 selection inputs, and the output enable (OE) signal (which is not present in this case). So, having excluded the output as a possible cellgroup, what is left?

A more suitable approach for a multiplexer would be to group all bits of an input and then create cellgroups of each input to grow backwards.

```
module MUX8to1 (input [32:0] a,     // 8-bit input
                input [32:0] b,     // 8-bit input
                input [32:0] c,     // 8-bit input
                input [32:0] d,     // 8-bit input
                input [32:0] e,     // 8-bit input
                input [32:0] f,     // 8-bit input
                input [32:0] g,     // 8-bit input
                input [32:0] h,     // 8-bit input
                input [2:0] sel,    // input sel used to select between a
    ,b,c,d
                output reg [32:0] out);   // 8-bit output based on input
     sel

   always @ (a or b or c or d or sel) begin
      case (sel)
         3'b000 : out <= a;
         3'b001 : out <= b;
         3'b010 : out <= c;
         3'b011 : out <= d;
         3'b100 : out <= e;
         3'b101 : out <= f;
         3'b110 : out <= g;
         3'b111 : out <= h;
      endcase
   end
endmodule
```

Listing 5.8: 33-bit 8 to 1 MUX

After synthesizing the RTL implementation of the MUX in the listing 5.8, the eight inputs a, b, c, d, e, f, g and h are clear. So, based on them, the flow takes the form below:

1. For input **a**:
   (a) For bit n, find the forward logic cone of input a (FLC_{an})
   (b) For the forward logic cone obtained, find and delete the output
   (c) For the forward logic cone obtained, find and delete the gates connected to the output
   (d) Add the remaining gates to the result list
2. For input **b**:
   (a) For bit n, find the forward logic cone of input b (FLC_{bn})
   (b) For the forward logic cone obtained, find and delete the output
   (c) For the forward logic cone obtained, find and delete the gates connected to the output
   (d) Add the remaining gates to the result list
3. For input **c**:
   (a) For bit n, find the forward logic cone of input c (FLC_{cn})
   (b) For the forward logic cone obtained, find and delete the output
   (c) For the forward logic cone obtained, find and delete the gates connected to the output
   (d) Add the remaining gates to the result list
4. inputs **d** to **g**...
5. For input **h**:
   (a) For bit n, find the forward logic cone of input h (FLC_{hn})
   (b) For the forward logic cone obtained, find and delete the output
   (c) For the forward logic cone obtained, find and delete the gates connected to the output
   (d) Add the remaining gates to the result list

Each result list, containing the gates of each input, will be a cellgroup.

Every step of the flow has been developed through TCL [8] functions and scripts that are being sourced in ASP tool [7].

```
set SEL 3
set INPUTS 8
set BITS 33

set result {}
set n 0

```

```
// collect gates for input a //
for {set i 0} {$i < $BITS} {incr i} {
  // step 1a //
  set input($i) [get_forward_logic_cone MUX8to1_k0/a|$i -longest]
  // step 1b //
  set noout($i) [delete_output $input($i)]

  // step 1c //
  set noout($i) [delete_output_gates $noout($i)]
  // step 1d //
  set result [concat $result $noout($i)]
}
set cellgroup($n) [extract_pins_from_conesList $result]

// reinitialise result list //
set result {}

incr n

// collect gates for input b //
for {set i 0} {$i < $BITS} {incr i} {
  set input($i) [get_forward_logic_cone MUX8to1_k0/b|$i -longest]
  set noout($i) [delete_output $input($i)]

  set noout($i) [delete_output_gates $noout($i)]
  set result [concat $result $noout($i)]
}
set cellgroup($n) [extract_pins_from_conesList $result]

// reinitialise result list //
set result {}

incr n

// collect gates for input c //
for {set i 0} {$i < $BITS} {incr i} {
  set input($i) [get_forward_logic_cone MUX8to1_k0/c|$i -longest]
  set noout($i) [delete_output $input($i)]

  set noout($i) [delete_output_gates $noout($i)]
  set result [concat $result $noout($i)]
}
set cellgroup($n) [extract_pins_from_conesList $result]

// reinitialise result list //
set result {}

// ..... inputs d-g ..... //

incr n

// collect gates for input h //
for {set i 0} {$i < $BITS} {incr i} {
  set input($i) [get_forward_logic_cone MUX8to1_k0/h|$i -longest]
  set noout($i) [delete_output $input($i)]

  set noout($i) [delete_output_gates $noout($i)]
  set result [concat $result $noout($i)]
}
set cellgroup($n) [extract_pins_from_conesList $result]
```

Listing 5.9: MUX Input Gates Extractions

Command used in steps #a, get_forward_logic_cone <pin_name> -longest, as mentioned in section 5.2.1, already exist in the ASP tool, and returns a list with the pins of the forward logic cone of pin_name. Similarly, command *concat*, exists already through the TCL interface and command *extract_pins_from_conesList*, was explained in listing 5.4.

Moving on to step #b, command *delete_output* does exactly what its name says, deletes the output pin:

```
proc delete_output {list} {
  set result {}

  foreach x $list {
    set substring "out"
    if {[string first $substring $x] == -1} {
      lappend result $x
    }
  }
  return $result
}
```

Listing 5.10: Procedure delete_output

Finally, command *delete_output_gates*, clarifies the list by removing the cells connected to the output. The procedure is in the listing 5.11.

```
// (listsize - 2) for 8 bits //
// (listsize - 6) for 33 bits (6 is for PINS, that is 3 GATES) //
// This number depends on the lib cells and their number of inputs //

proc delete_output_gates {list} {
  set result {}
  set listsize [llength $list]

  set i 0
  foreach x $list {
    if {$i < [expr $listsize - 6] } {
      lappend result $x
    }
    incr i
  }
  return $result
}
```

Listing 5.11: Procedure delete_output_gates

Basically, this process creates cellgroups of all the bits of each input. For example, for the fig. 5.4, an 8 to 1 **1 bit** MUX, that is, each input has 1 bit, the cellgroups would be eight with only 1 entry each of them.

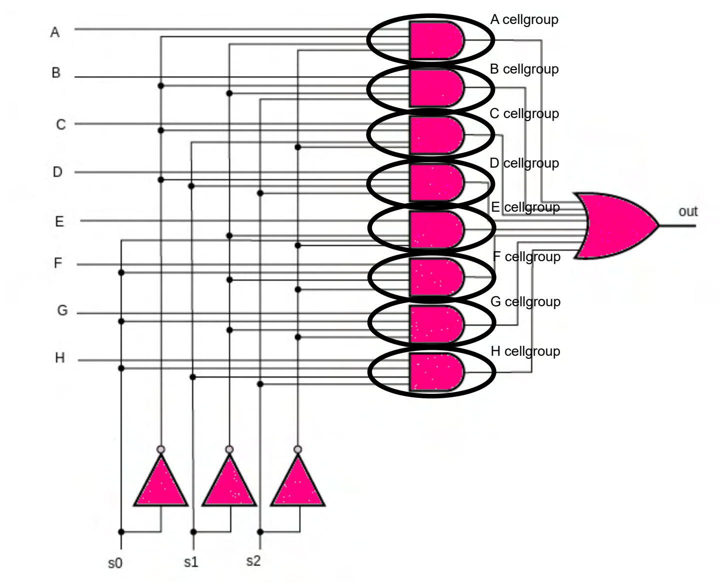


Figure 5.5: Cellgroups of 1-bit 8 to 1 Multiplexer

So, after synthesizing the listing 5.8, following the flow, and loading the netlist into the ASP tool [7], the regularity measurements are:

```
> INFO: Total Cells = 415, Total Grouped Cells = 264, Grouped Cell Ratio = 64%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 12079.87um^2,
> Total Grouped Cells Area = 7451.14um^2, Grouped Cell Area Ratio = 62%
```

For the specific circuit, another possible action would be to create a cellgroup of all the bits of the output cells and grow it backwards. In this way, each bit will grow backwards and will manage to group the "output" gates that were excluded from the flow before. Although, this is a technique characterized "design-specific", while the flow in the listing 5.9 could be applied in more designs with similar behavior as MUX, that is not expectable logic depth.

The output cellgroup that could be created is:

```
create_cellgroup -module MUX8to1 -groupid 1 -groupname OUT { MUX8to1/
   U416 MUX8to1/U428 MUX8to1/U440 MUX8to1/U452 MUX8to1/U464 MUX8to1/U476
    MUX8to1/U488 MUX8to1/U500 MUX8to1/U512 MUX8to1/U524 MUX8to1/U536
   MUX8to1/U548 MUX8to1/U560 MUX8to1/U572 MUX8to1/U584 MUX8to1/U596
   MUX8to1/U608 MUX8to1/U620 MUX8to1/U632 MUX8to1/U644 MUX8to1/U656
   MUX8to1/U668 MUX8to1/U680 MUX8to1/U692 MUX8to1/U704 MUX8to1/U716
   MUX8to1/U728 MUX8to1/U740 MUX8to1/U752 MUX8to1/U764 MUX8to1/U776
   MUX8to1/U788 MUX8to1/U800}
```

Listing 5.12: MUX "output" cellgroup

After growing it backwards, with the isomorphism algorithm, the regularity measurement is:

```
> INFO: Total Cells = 415, Total Grouped Cells = 396, Grouped Cell Ratio = 95%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 12079.87um^2,
> Total Grouped Cells Area = 11642.40um^2, Grouped Cell Area Ratio = 96%
```

Chapter 6

Bit Slicing Regular Datapath Placement Flow

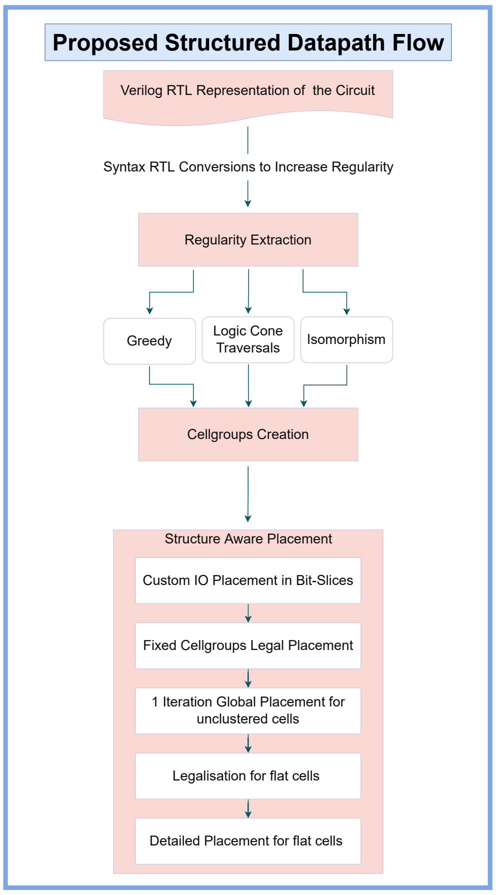

Proposed Structured Datapath Flow
Verilog RTL Representation of the Circuit
Syntax RTL Conversions to Increase Regularity
Regularity Extraction
Greedy
Logic Cone Traversals
Isomorphism
Cellgroups Creation
Structure Aware Placement
Custom IO Placement in Bit-Slices
Fixed Cellgroups Legal Placement
1 Iteration Global Placement for unclustered cells
Legalisation for flat cells
Detailed Placement for flat cells

Figure 6.1: Structured Datapath Placement Proposed Flow

The proposed SDP flow consists of several key steps to achieve efficient and accurate placement of electronic circuits. Firstly, regularity is extracted using one of three methods:

- the greedy method,
- the isomorphism method,

- the proposed method using logic cones.

Once regularity is established, cellgroups are created and placement is carried out. The IO placement step involves either custom or automated methods, depending on the design requirements. Cellgroups are then fixed in legal positions, and a one-iteration global placement is performed for flat cells. Legalization is then carried out to ensure that flat cells are placed within legal boundaries. Finally, detailed placement is executed for flat cells, completing the placement process. By following this proposed SDP flow, efficient and accurate placement of electronic circuits can be achieved, allowing for optimal performance and functionality.

Normally, detailed placement consumes a great amount of time due to its complexity. In this flow, because it is only performed for flat cells, it is significantly fast, while at the same time better results are achieved.

6.1 Case Studies & their Hierarchical Break-Down Analysis

The case studies of this report are below:

- Register Files
  - 32x32
  - 64x64
- 32x32 Booth Multiplier
  - Synthetic Adder
    - 33-bit
    - 64-bit
  - MUX Tree (x16 MUX)
  - CSA Tree (x16 CSA)
- DLX Execute Unit

Register files were chosen because of their natural structured behavior to be the first case study and prove the theory. *Booth Multiplier* and *DLX's Execute Unit* were chosen due to their massive use from GPUs to microprocessors.

6.2 Regular Placement of Bit-Slices

6.2.1 Register Files

The system being analyzed first, stores and manipulates data using registers organized in rows and columns. It is regular in structure by nature, making it easily expandable. To achieve this, the proposed flow includes creating cell groups of registers, using the **2**nd *proposed method of the flow step*, ordered from 0 to 31, which helps extract regularity and reach up to 73%. These cell groups are then fixed placed manually using ECO commands provided by the ASP tool. The flat logic, which makes up 27% of the system, is then addressed through a single iteration global placement, legalization and finally, detailed placement.

32x32 RF

The regularity measured in ASP tool [7], after the cellgroup creation is:

```
> INFO: Total Cells = 5660, Total Grouped Cells = 4160, Grouped Cell Ratio = 73%
> INFO: Clustering Complete.
> INFO: Verifying Clustering Result.
> INFO: Clustering Result Verified OK.
> INFO: Top-Level Module Area = 263788.56um^2,
> Total Grouped Cells Area = 217240.13um^2, Grouped Cell Area Ratio = 83%
```

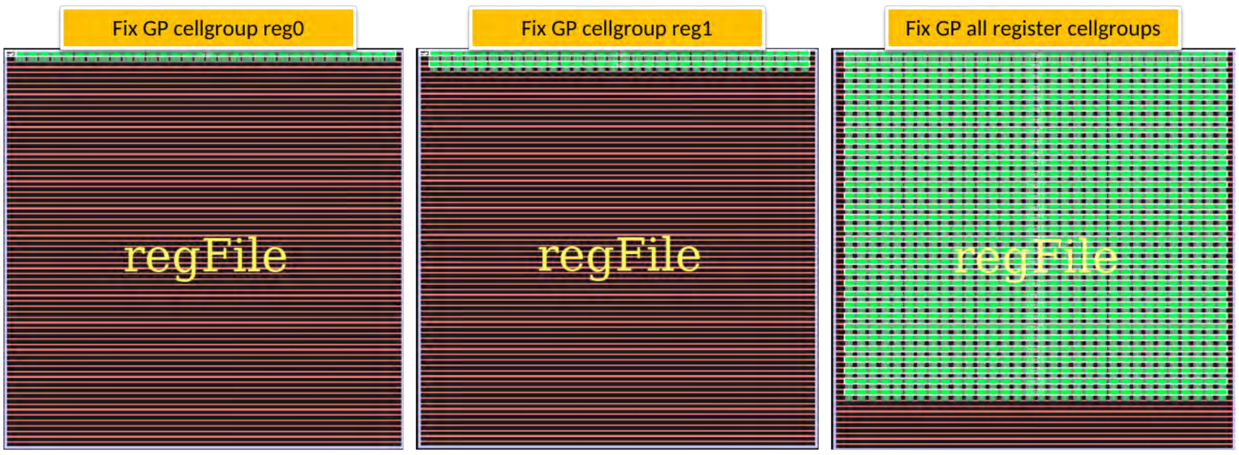


Figure 6.2: Fix Place Cell-groups row-by-row

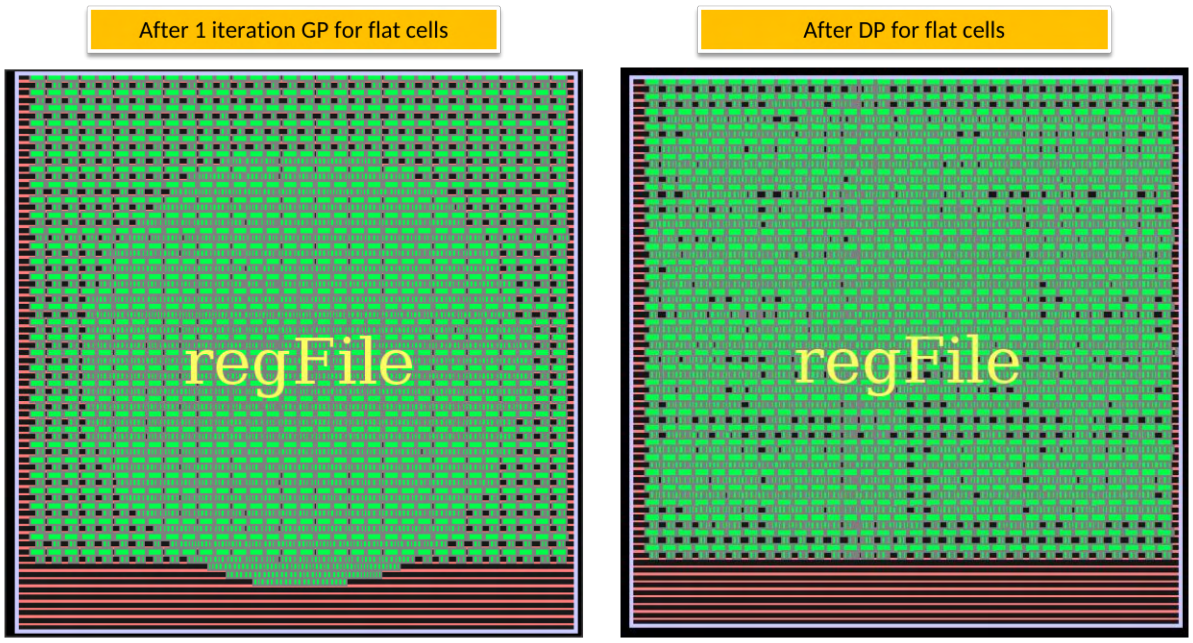


Figure 6.3: GP/LG/DP of 32x32 RegFile's Flat Cells

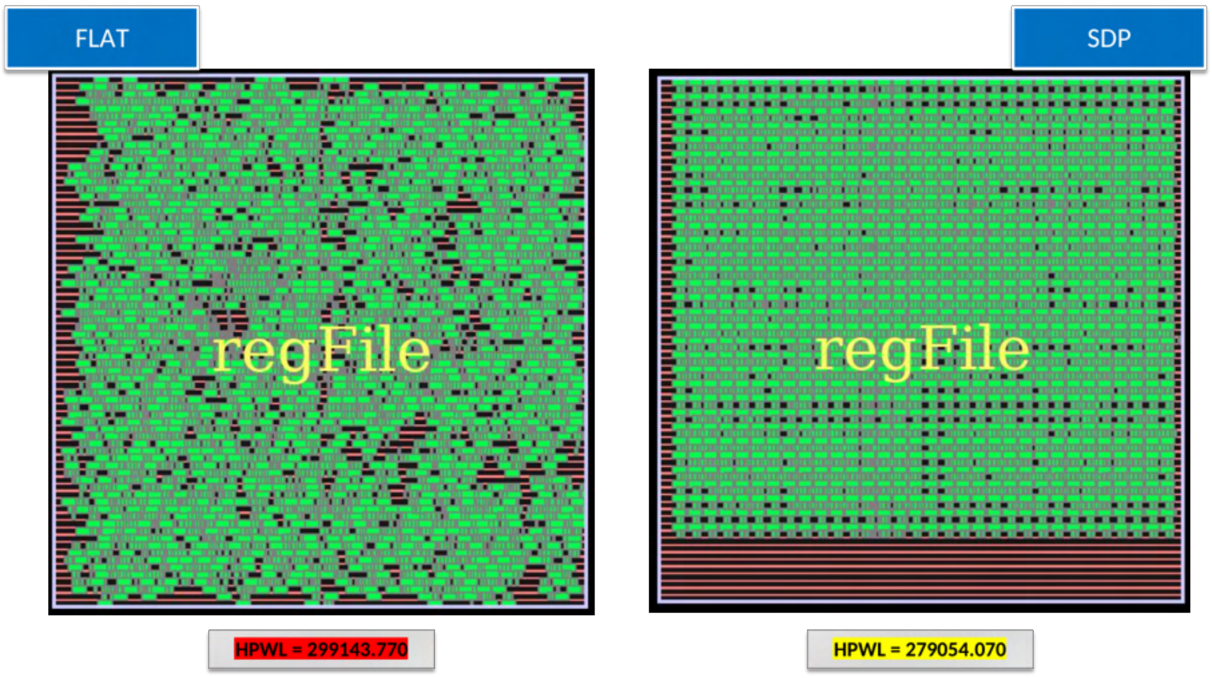


Figure 6.4: Flat VS SDP for 32x32 RegFile

As shown in fig. 6.4, the area and the half perimeter wirelength for the **SDP** version compared to the **flat** one is decreasing. More specifically, the ***area improvement* is 10%** and the ***HPWL improvement* is 7%**.

64x64 RF

The regularity extraction process, along with the ECO manual placement of the cellgroups for the 64x64 Register File, follows the same approach with the 32x32. In fig. 6.5, it is shown how **packed** the cells are, leading to **reduced area** and **reduced total wirelength**.

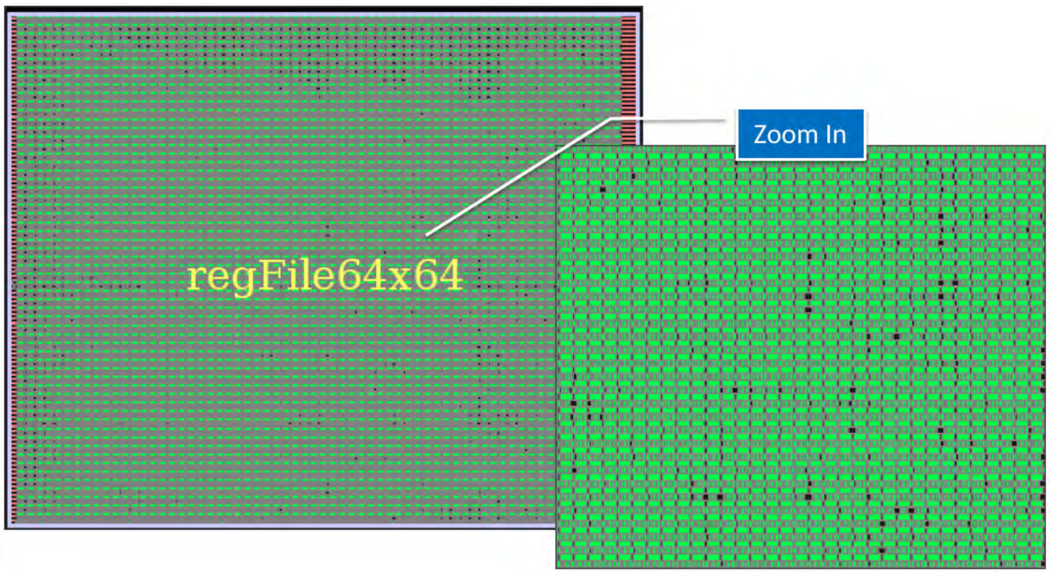


Figure 6.5: RegFile 64x64

6.2.2 32x32 Booth Multiplier

Multipliers are commonly used in a variety of computing systems, ranging from GPUs (Graphics Processing Units) to microprocessors. Multipliers are electronic circuits that perform the arithmetic operation of multiplication, which is a fundamental operation in many computational tasks. In GPUs, multipliers are used to perform matrix multiplication, which is a key operation in graphics rendering and machine learning tasks. Multipliers can be implemented using different techniques such as combinational circuits, sequential circuits, or pipelined circuits. Depending on the specific application and performance requirements, different types of multipliers may be used. For example, high-performance microprocessors may use pipelined multipliers that can perform multiple multiplication operations in parallel to improve processing speed.

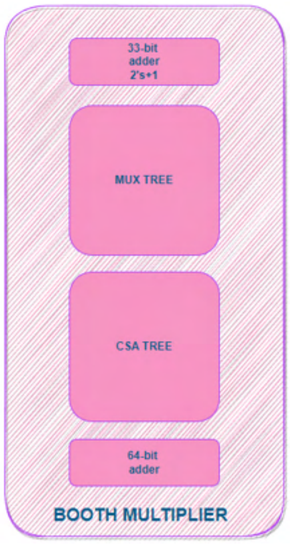


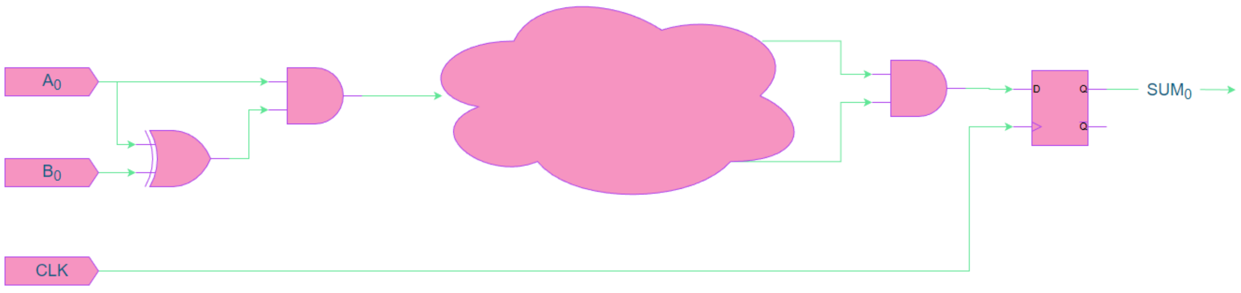
Figure 6.6: Booth Multiplier Block Diagram

The Booth multiplier is one of the first types of multipliers to be analyzed in general, but the only one in terms of this report, and it consists of four basic blocks. The two first blocks are two adders, the first one is used to calculate the two's complement of the multiplier and the second one, to add the two last outputs from the carry-save adder (CSA) to obtain the final result. The third block is a tree of 16 multiplexers (MUXs), which are used to prepare the inputs for the CSA. Finally, the fourth block is a tree of 16 64-bit CSAs that add the partial products to calculate the final result. The Booth multiplier is an efficient design for multiplying two signed binary numbers, as it reduces the number of partial products required for the multiplication operation, resulting in faster computation times and reduced hardware complexity. Overall, the Booth multiplier is a commonly used and important component in digital systems, particularly in microprocessors and other integrated circuits.

Block by block, each one is going to be analyzed and placed in a structured way to try and achieve better results. The first one will be the 33-bit synthesized adder.

33x33 Synthetic Adder

The proposed flow for regularity extraction involves using ***the method with logic cones proposed in the previous chapter***. As far as the creation of the cellgroups is concerned, this involves dividing the SUM_n signal into two groups. The first group, called the *input group*, consists of the cells from the first two levels of the SUM_n logic cone. The second group, called the *non-input group*, consists of the remaining cells from levels [length-2] of the SUM_n logic cone. By grouping these cells together based on their logical relationships, we can identify regular patterns in the circuit that can be exploited to optimize the design. This process can be used to identify areas of the circuit that are redundant or could be simplified, leading to improved performance and reduced hardware complexity. Regularity extraction using logic cones is an important technique in digital design and can be applied to a wide range of circuits, including adders and other arithmetic circuits.



Regarding the placement flow, it starts with the IO placement. The placement of input and output (IO) pins is an important aspect of digital design, as it can affect the performance, reliability, and manufacturability of the circuit. In the proposed flow for IO placement, the IOs are placed on the west and east side of the core area, respectively, bit by bit, as shown in the fig. 6.7.

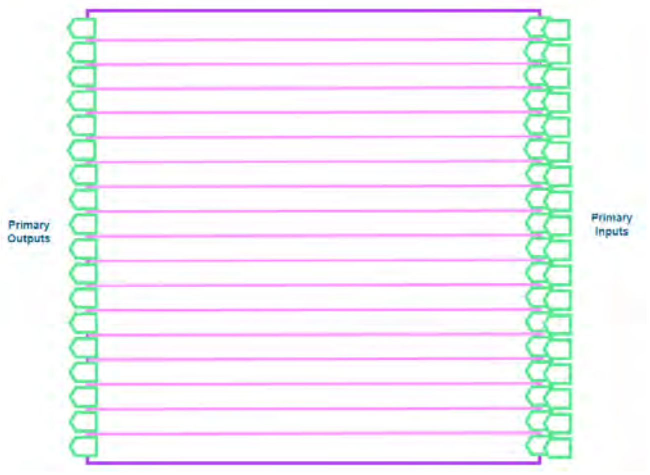


Figure 6.7: Proposed IOs for 33-bit Synthetic Adder

The placement flow consists of several steps. First, the input group is placed row by row on the east side of the core, depending on where the IOs have been placed. This placement is fixed and cannot be changed. Next, the non-input group is placed row by row, leaving space for the carry chain, which is used to propagate the carry bit in multi-bit arithmetic operations.

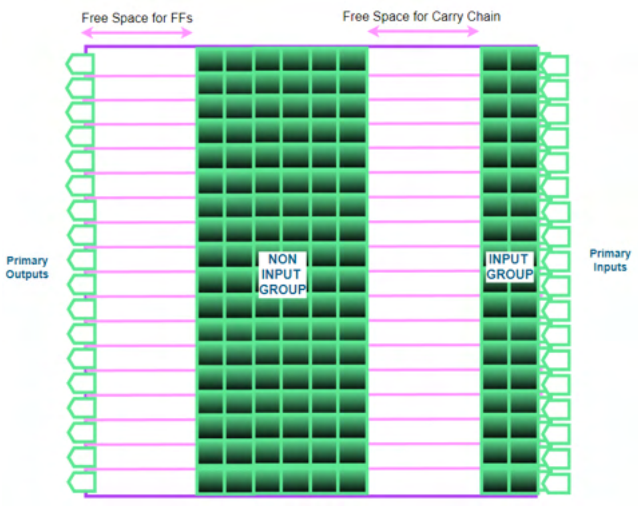


Figure 6.8: Adder's Cellgroups Fixed Placement

After the initial placement is complete, a global placement iteration is performed to optimize the placement of the cells. The placement is then legalized, which ensures that the cells are placed within the legal constraints of the design rules. Finally, a detailed placement step is performed, which further refines the placement of the cells to optimize performance, minimize power consumption, and reduce the area of the circuit.

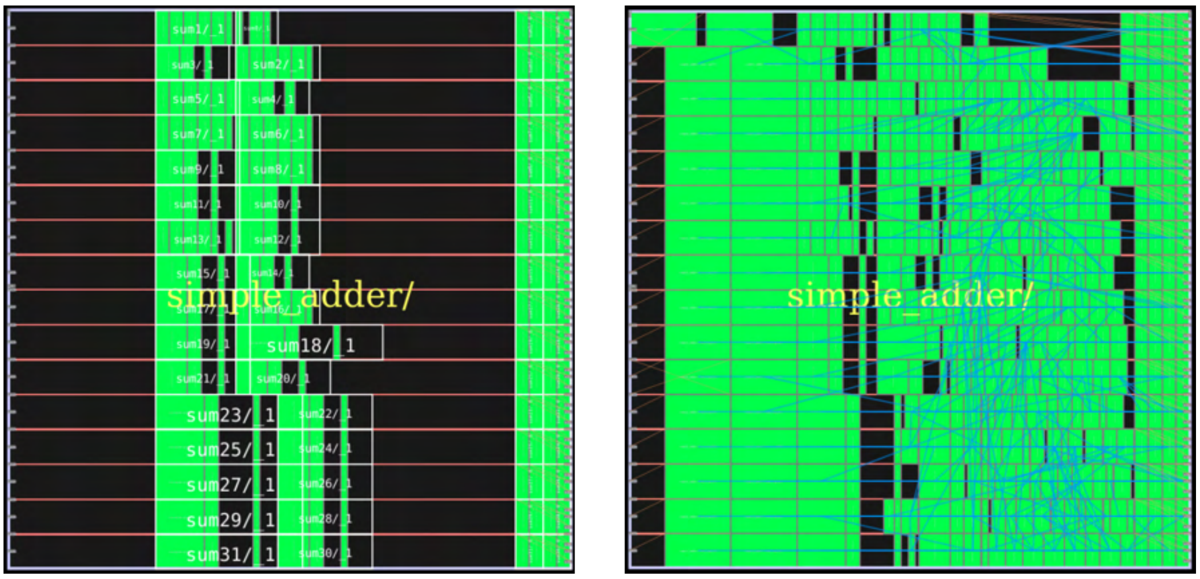
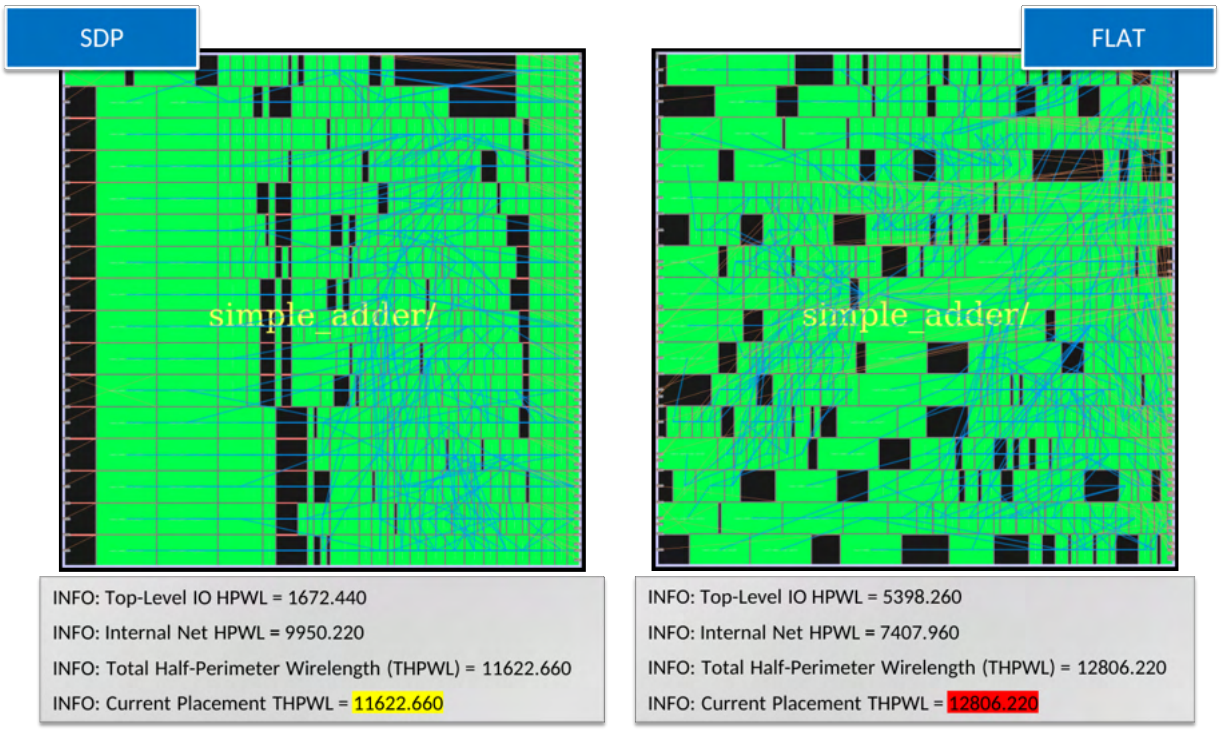


Figure 6.9: Fix Place Sum Cellgroups & GP/LG/DP for Flat Ones



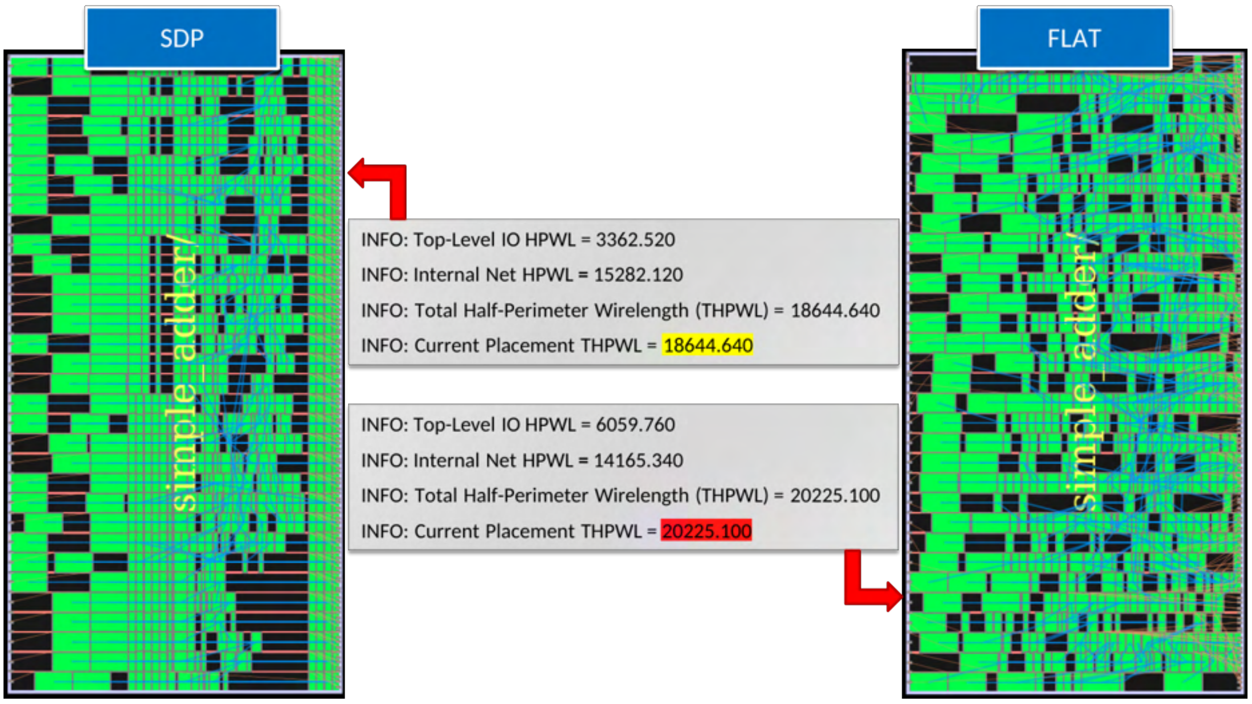
```
INFO: Top-Level IO HPWL = 1672.440
INFO: Internal Net HPWL = 9950.220
INFO: Total Half-Perimeter Wirelength (THPWL) = 11622.660
INFO: Current Placement THPWL = 11622.660
```

```
INFO: Top-Level IO HPWL = 5398.260
INFO: Internal Net HPWL = 7407.960
INFO: Total Half-Perimeter Wirelength (THPWL) = 12806.220
INFO: Current Placement THPWL = 12806.220
```

Figure 6.10: Flat VS SDP for 33-bit Synthetic Adder

As shown in fig. 6.10, the half perimeter wirelength for the **SDP** version compared to the **flat** one is decreasing. More specifically, the ***improvement* is 9.24%**.

64x64 Synthetic Adder



```
INFO: Top-Level IO HPWL = 3362.520
INFO: Internal Net HPWL = 15282.120
INFO: Total Half-Perimeter Wirelength (THPWL) = 18644.640
INFO: Current Placement THPWL = 18644.640
```

```
INFO: Top-Level IO HPWL = 6059.760
INFO: Internal Net HPWL = 14165.340
INFO: Total Half-Perimeter Wirelength (THPWL) = 20225.100
INFO: Current Placement THPWL = 20225.100
```

Figure 6.11: Flat VS SDP for 64-bit Synthetic Adder

The proposed flow for the 64-bit synthetic adder is exactly the same, so the results are in fig. 6.11. The HPWL improvement is 7.81%.

MUX

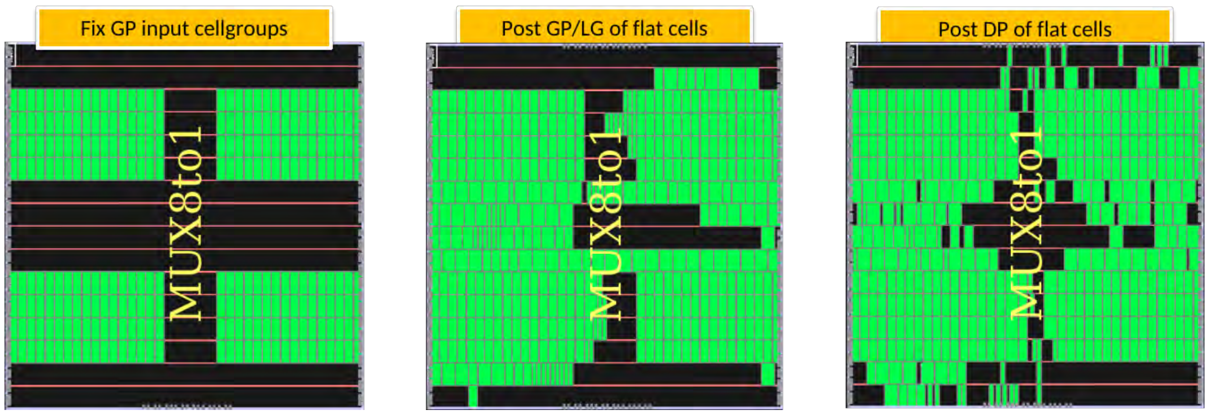
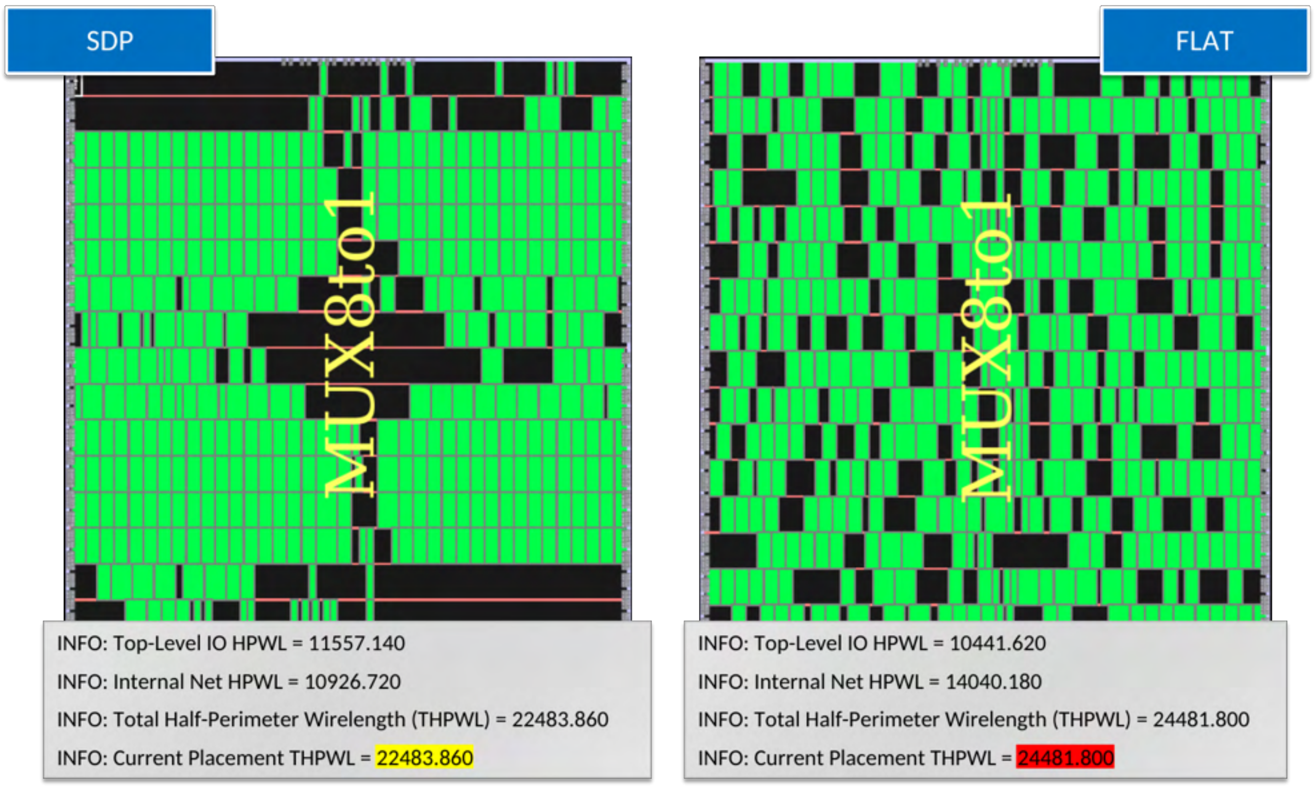


Figure 6.12: Fix Place Input Cell-groups & GP/LG/DP for Flat Cells

The second block is the MUX tree, a tree of 16 33-bit 8 to 1 MUX. The proposed flow refers to only the one of them and involves using ***the logic cone method, proposed in the previous chapter***, to extract regularity and create cell groups. The cell groups are then manually fixed in place using an engineering change order (ECO) process. After the cell groups are fixed in place, the flat logic is placed using GP/LG/DP for the select and output signals. This process may help to simplify the design of the MUX tree and improve its overall efficiency and functionality.



```
INFO: Top-Level IO HPWL = 11557.140
INFO: Internal Net HPWL = 10926.720
INFO: Total Half-Perimeter Wirelength (THPWL) = 22483.860
INFO: Current Placement THPWL = 22483.860
```

```
INFO: Top-Level IO HPWL = 10441.620
INFO: Internal Net HPWL = 14040.180
INFO: Total Half-Perimeter Wirelength (THPWL) = 24481.800
INFO: Current Placement THPWL = 24481.800
```

Figure 6.13: Flat VS SDP for 33-bit 8 to 1 Synthetic MUX

As shown in fig. 6.13, the half perimeter wirelength for the **SDP** version compared to the **flat** one is decreasing. More specifically, the ***improvement*** **is 8.16%**.

CSA Core - 16xCSA

The proposed flow for the tree of 16 CSAs involves several steps. First, regularity extraction and cell group creation will be done using the 1st ***and*** 2nd ***proposed methods of the RTL techniques flow***. Then, the primary inputs are placed, with the first half on top (south of the core) and the other half on the bottom (north of the core). Next, the placement of register cell groups will be done manually, with multiple combinations based on their connections. In this report, only one of those combinations will be shown, the one in fig. 6.14, but there were over 100 of them tried. The process of finding the best one is hard and complex, and one of the main projects of the *future work*.

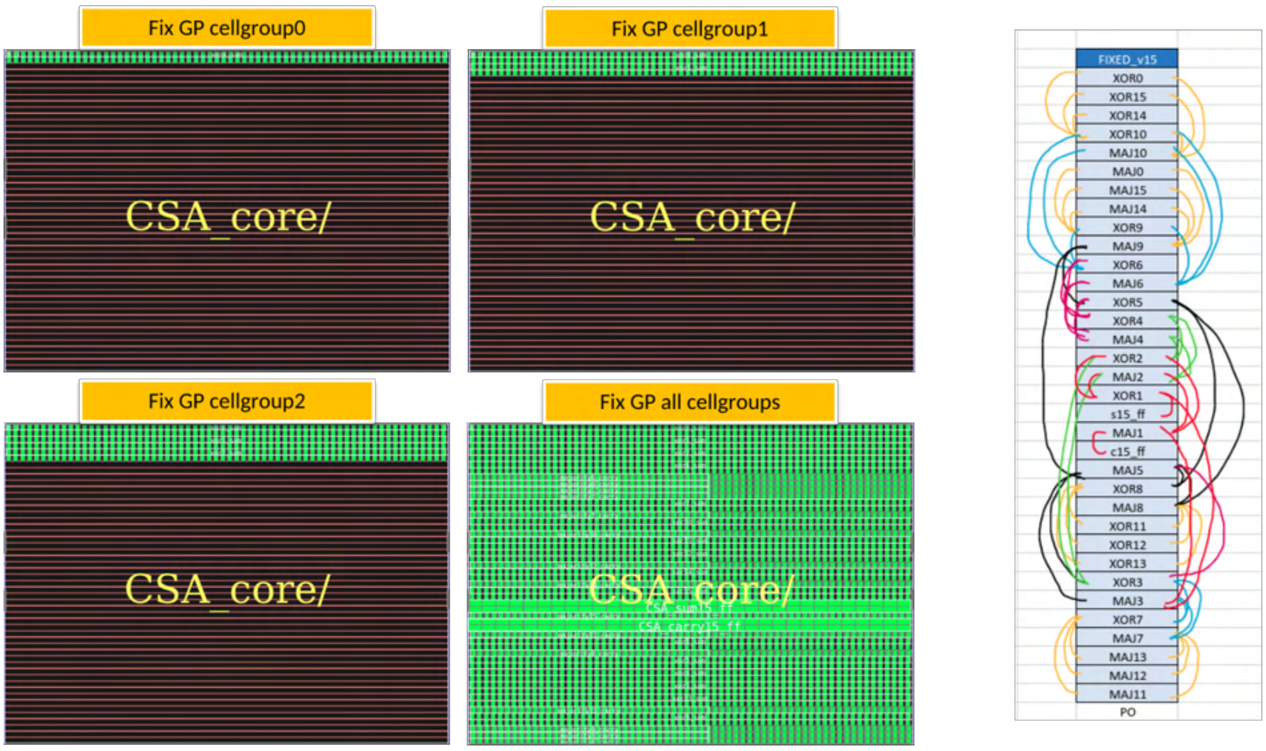
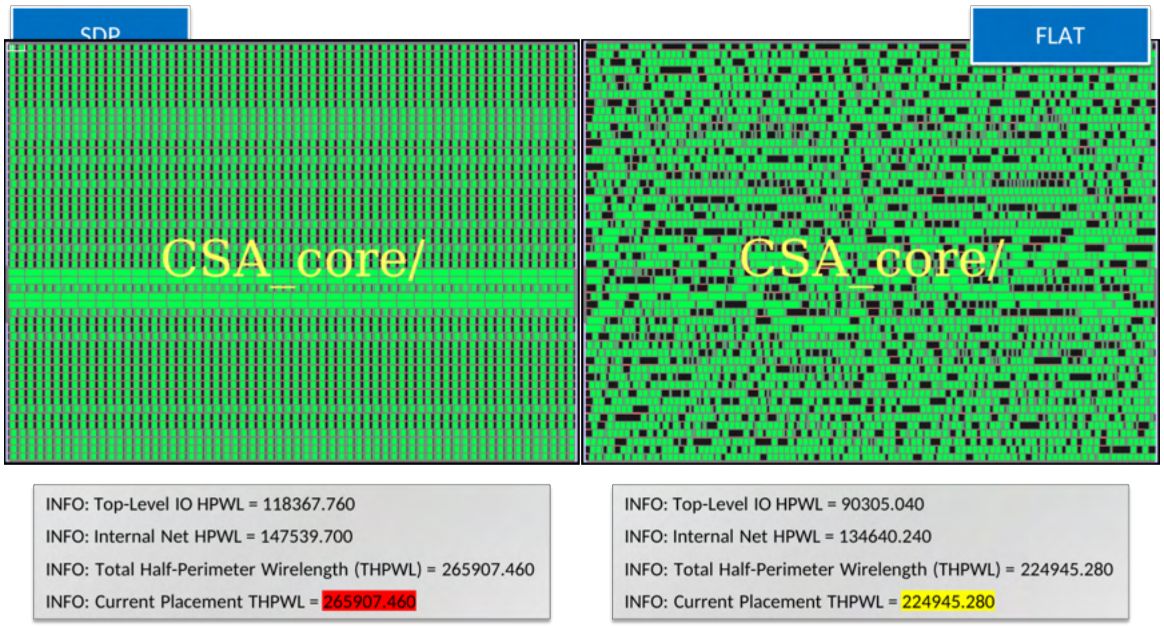


Figure 6.14: Cell-groups Fixed Placement for the Right Combination

Finally, flat cells will be placed using 1 iteration of global placement, local placement, and detailed placement. This process aims to create an efficient and organized tree of 16 CSAs with a 64-bit structure, while also ensuring the connections between cells are optimized for optimal functionality.

In fig. 6.15, there is the first attempt of cellgroups' placement with the combinations shown in fig. 6.14. As it seems, the HPWL in the *flat* placement is lower than the *SDP* one. This may be happening for a lot of reasons, with the first and most likely one to be that the combination is not optimal. Another one maybe the primary inputs/outputs placement. This placement results in **18.2%** HPWL worsening.

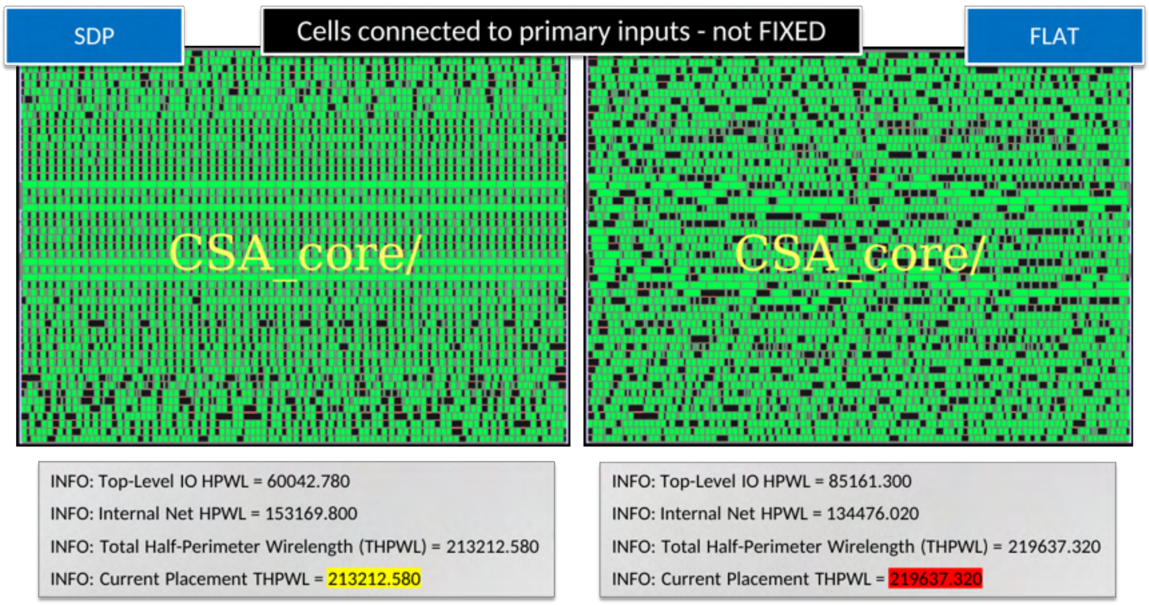


```
INFO: Top-Level IO HPWL = 118367.760
INFO: Internal Net HPWL = 147539.700
INFO: Total Half-Perimeter Wirelength (THPWL) = 265907.460
INFO: Current Placement THPWL = 265907.460
```

```
INFO: Top-Level IO HPWL = 90305.040
INFO: Internal Net HPWL = 134640.240
INFO: Total Half-Perimeter Wirelength (THPWL) = 224945.280
INFO: Current Placement THPWL = 224945.280
```

Figure 6.15: Flat VS SDP for CSA Tree (Attempt 1)

By examining the measurements, the **IO HPWL** is approximately 30000 more in the SDP version, while the **Internal HPWL** is around 12000 more in the **SDP**. So, another approach would be to try and decrease the *IO HPWL*, by leaving the cellgroups connected to the primary inputs, to be treated as flat cells. By doing that, the IOs will pull closer to them the cells that they connect, leading to lower *IO HPWL*. The results are in fig. 6.16, where IO HPWL is approximately 25000 lower in **SDP** than in the **flat** one. At the same time, the *internal HPWL* is approximately 19000 more in the **SDP** than in the **flat**, leading to results very close to one another. The total HPWL improvement here, though, is ***3%***.



```
INFO: Top-Level IO HPWL = 60042.780
INFO: Internal Net HPWL = 153169.800
INFO: Total Half-Perimeter Wirelength (THPWL) = 213212.580
INFO: Current Placement THPWL = 213212.580
```

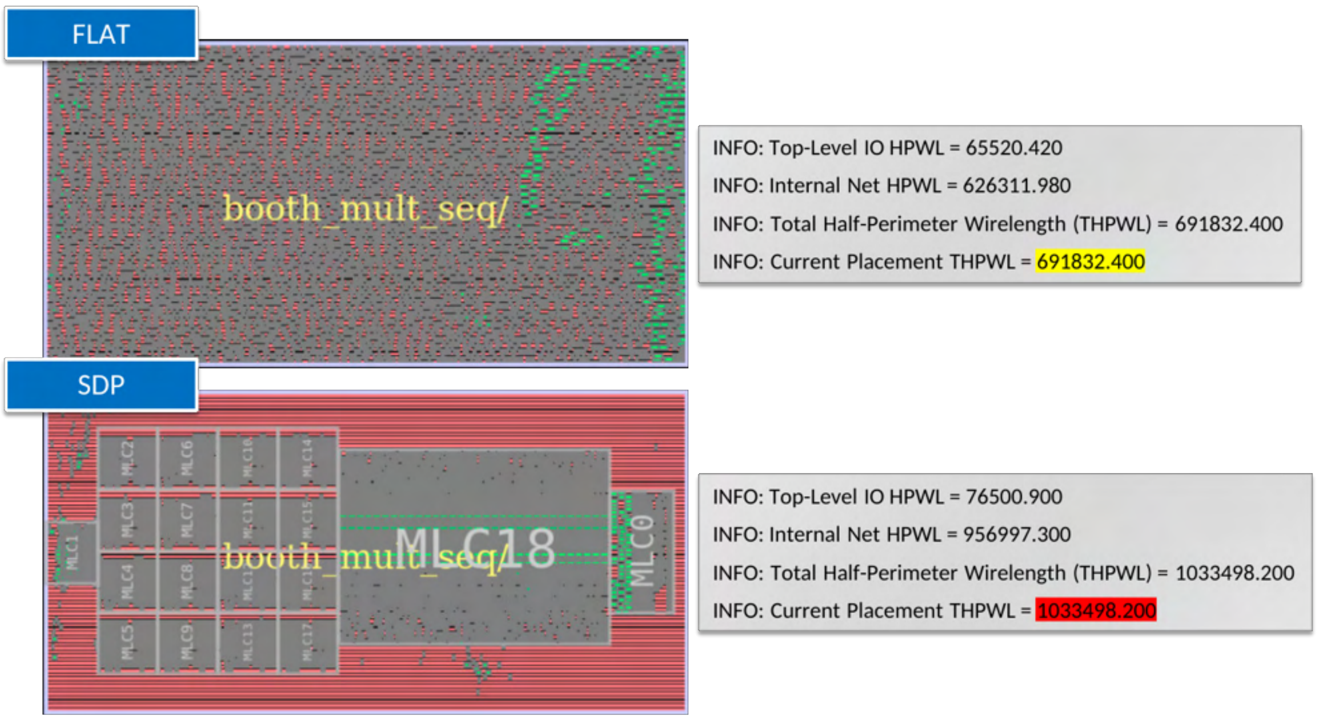
```
INFO: Top-Level IO HPWL = 85161.300
INFO: Internal Net HPWL = 134476.020
INFO: Total Half-Perimeter Wirelength (THPWL) = 219637.320
INFO: Current Placement THPWL = 219637.320
```

Figure 6.16: Flat VS *Semi-SDP* for CSA Tree (Attempt 2)

Booth Multiplier - Combining Previous Modules

Finally, after examining and analyzing every module separately, the time has come to combine all of them. The proposed flow here involves *ml clusters* and more specifically creating several of them, for different components, namely the 33-bit adder, all 16 MUXs, the whole CSA core, and the 64-bit adder. Once the clusters are created, the next step is to place them on a global ML grid and fill them in with their relative positions. This process is essential to ensure that the clusters are positioned correctly and do not overlap with one another. Finally, the clusters' placement is legalized, which means that any potential violations of the design rules are identified and resolved. This process ensures that the design is functional and can be fabricated correctly.

In fig. 6.17, there is the first attempt of the flow. The primary inputs are all placed on the west of the core, while the primary outputs are on the east. *The way the ml clusters are positioning here, is custom and designed regarding the booth structure.*



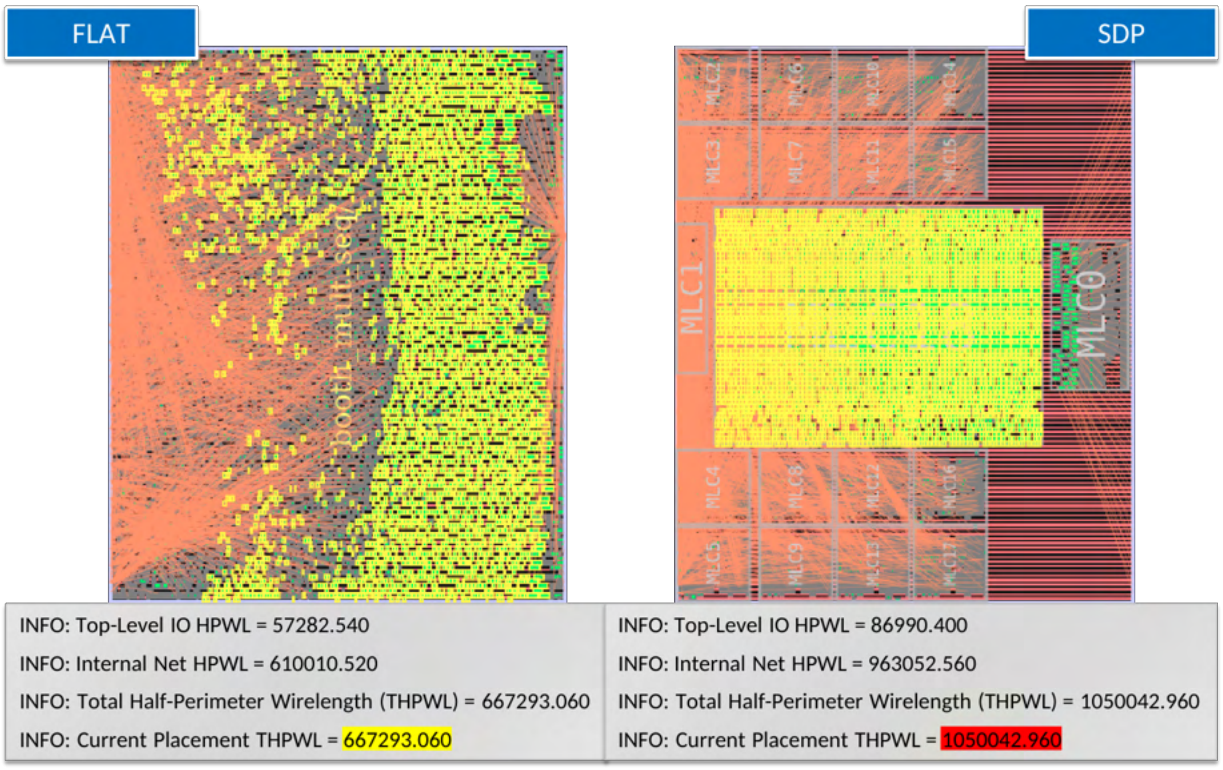
```
INFO: Top-Level IO HPWL = 65520.420
INFO: Internal Net HPWL = 626311.980
INFO: Total Half-Perimeter Wirelength (THPWL) = 691832.400
INFO: Current Placement THPWL = 691832.400
```

```
INFO: Top-Level IO HPWL = 76500.900
INFO: Internal Net HPWL = 956997.300
INFO: Total Half-Perimeter Wirelength (THPWL) = 1033498.200
INFO: Current Placement THPWL = 1033498.200
```

Figure 6.17: Booth Multiplier Flat VS SDP (Attempt 1)

As shows in fig. 6.17, the SDP version is approximately 2x the flat one.

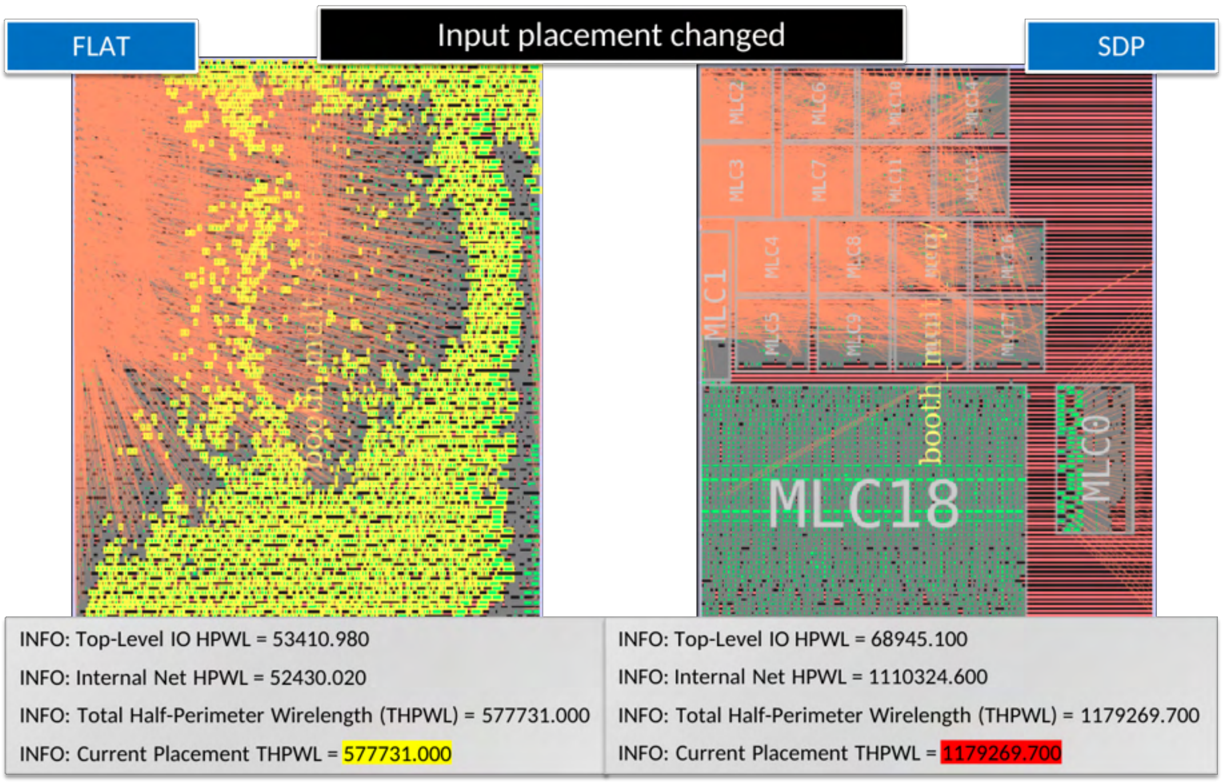
In fig. 6.18, on the other hand, a different approach is followed as far as the top level floorplan is concerned. The core box is rotated and the MUX clusters are placed in a way that they are closer to the CSA cellgroups they are connected to. Nonetheless, the HPWL of the SDP version is still more than the flat one.



```
INFO: Top-Level IO HPWL = 57282.540
INFO: Internal Net HPWL = 610010.520
INFO: Total Half-Perimeter Wirelength (THPWL) = 667293.060
INFO: Current Placement THPWL = 667293.060
```

```
INFO: Top-Level IO HPWL = 86990.400
INFO: Internal Net HPWL = 963052.560
INFO: Total Half-Perimeter Wirelength (THPWL) = 1050042.960
INFO: Current Placement THPWL = 1050042.960
```

Figure 6.18: Booth Multiplier Flat VS SDP (Attempt 2)



```
INFO: Top-Level IO HPWL = 53410.980
INFO: Internal Net HPWL = 52430.020
INFO: Total Half-Perimeter Wirelength (THPWL) = 577731.000
INFO: Current Placement THPWL = 577731.000
```

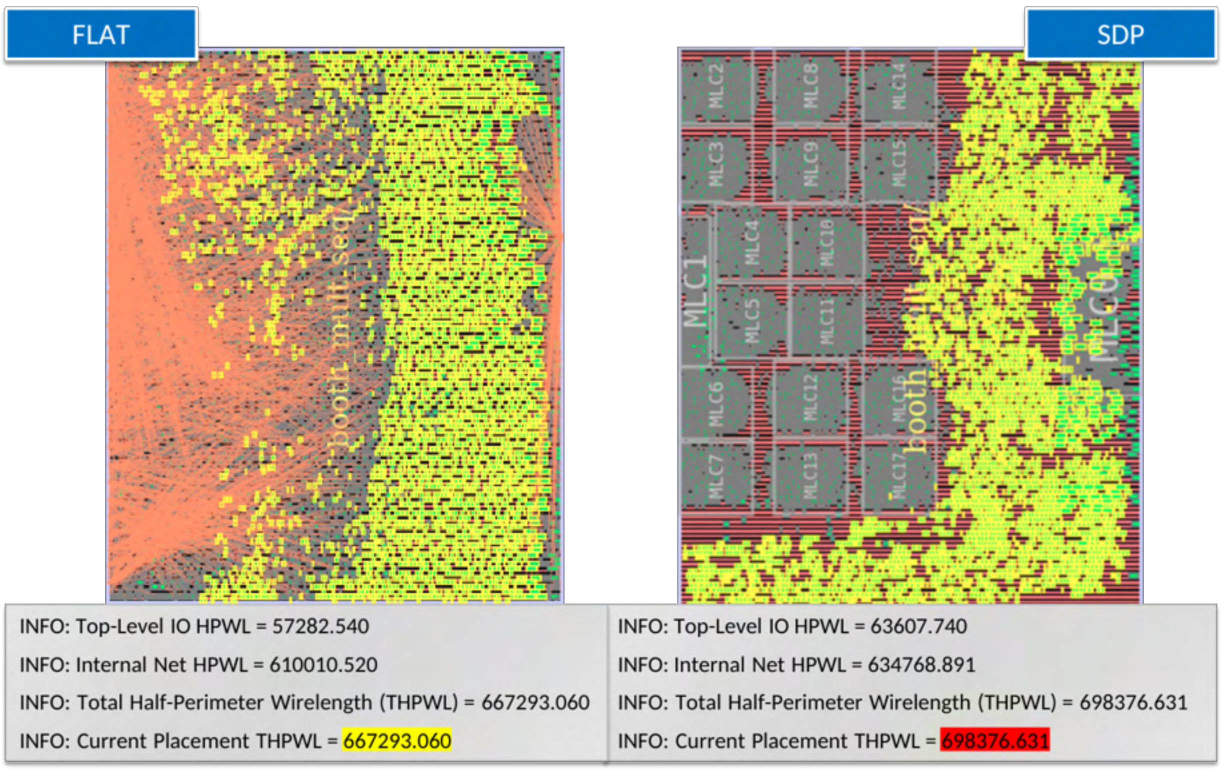
```
INFO: Top-Level IO HPWL = 68945.100
INFO: Internal Net HPWL = 1110324.600
INFO: Total Half-Perimeter Wirelength (THPWL) = 1179269.700
INFO: Current Placement THPWL = 1179269.700
```

Figure 6.19: Booth Multiplier Flat VS SDP (Attempt 3)

In fig. 6.19, there has been an attempt to reduce the IO HPWL by gathering in the top left corner all the primary inputs and, that is, the MUX clusters they are connected too. Still not the wanted results, though.

After all these unsuccessful attempts to place the clusters, the conclusion is one. The fact that each module is placed optimally for itself, with reduced HPWL connections, does not mean that the combination of them will be optimal too. The fact that the internal connections are reduced, creates a bigger possibility for the outer connections to be more dense, that is, the connections from cluster to cluster.

For the final attempt shown, the *CSA core* cluster (highlighted one) is flattened. This cluster is the more difficult to be placed due to its size and dimensions. By making it flat, it gives the placer the freedom to examine better positions for the CSA cells that are connected to the MUXs and could be near them, due to the restrictions of the box. As shown in fig. 6.20, the total HPWL is still more in the SDP version than in the flat one, but significantly better.



```
INFO: Top-Level IO HPWL = 57282.540
INFO: Internal Net HPWL = 610010.520
INFO: Total Half-Perimeter Wirelength (THPWL) = 667293.060
INFO: Current Placement THPWL = 667293.060
```

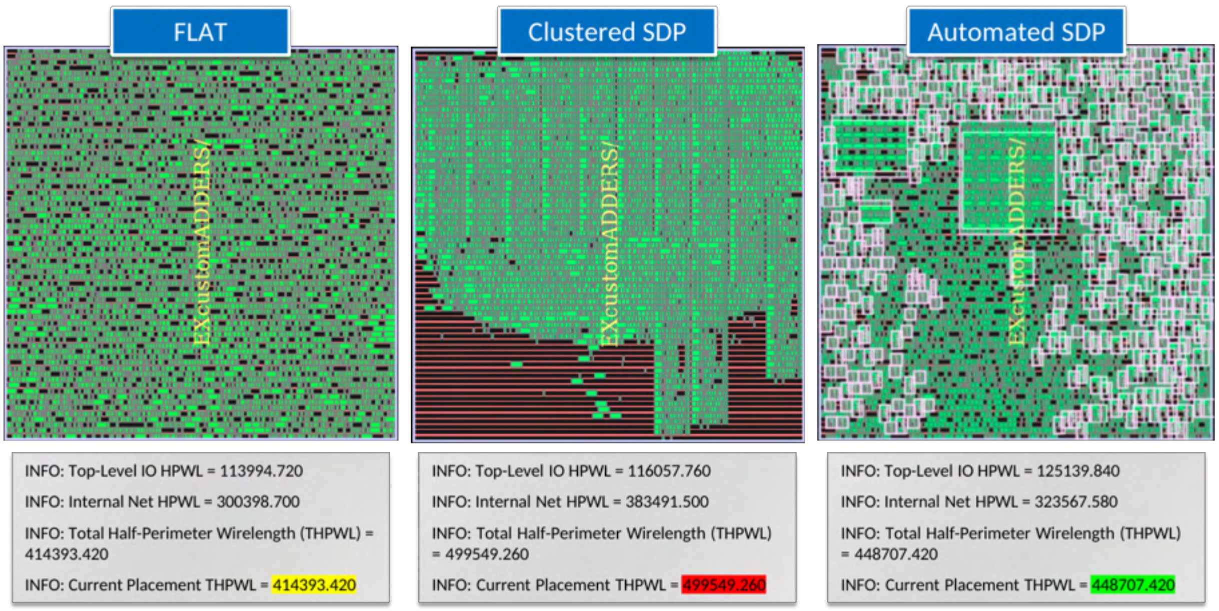
```
INFO: Top-Level IO HPWL = 63607.740
INFO: Internal Net HPWL = 634768.891
INFO: Total Half-Perimeter Wirelength (THPWL) = 698376.631
INFO: Current Placement THPWL = 698376.631
```

Figure 6.20: Booth Multiplier Flat VS SDP (Attempt 4)

6.2.3 DLX Execute Unit

For the last case study, the *DLX's Execute Unit*, the proposed flow involves several steps. Firstly, the regularity of the design is extracted, and cell groups are created using ***the third proposed method from the RTL changes flow***. This process ensures that the design is optimally structured and arranged for efficient placement. Secondly, the IO is placed using a default configuration, provided by the ASP tool. This step ensures that the input and output interfaces are correctly positioned for the design's functionality.

The next step is the placement phase, which could be done with two possible ways. The first one, involves placing the ML clusters using the same flow as with the Booth algorithm from before. This process ensures that the ML clusters are correctly positioned and connected for efficient operation. The second one is the automated ASP SDP placement. This process ensures fast results, without any custom action, while at the same time adheres to the design rules and any potential violations are identified and corrected.



```
INFO: Top-Level IO HPWL = 113994.720
INFO: Internal Net HPWL = 300398.700
INFO: Total Half-Perimeter Wirelength (THPWL) =
414393.420
INFO: Current Placement THPWL = 414393.420
```

```
INFO: Top-Level IO HPWL = 116057.760
INFO: Internal Net HPWL = 383491.500
INFO: Total Half-Perimeter Wirelength (THPWL)
= 499549.260
INFO: Current Placement THPWL = 499549.260
```

```
INFO: Top-Level IO HPWL = 125139.840
INFO: Internal Net HPWL = 323567.580
INFO: Total Half-Perimeter Wirelength (THPWL)
= 448707.420
INFO: Current Placement THPWL = 448707.420
```

Figure 6.21: Flat VS Clustered SDP ($1^{st}approach$) VS Automated SDP ($2^{nd}approach$)

In fig. 6.21 there is the comparison between the *flat version*, the *clustered SDP* and the *automated SDP*, ordered from better to worse.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

To conclude, in this thesis we proposed three flows. The first one consists of three techniques that could be applied in verilog RTL level and help increase regularity, namely module separation, cell-groups creations and finally reusage of already regular modules. The second one applies in circuits that the RTL implementation is not available, or the designer simply wants to let the synthesis tool maximize the optimizations and involves using logic cones to identify the cell-groups and extract the regularity. The last proposed flow is a structured data path one, and it combines regularity extraction, with either one of the structured clustering algorithms, or the logic cones flow proposed before, and structure aware placement, either custom or automated.

The results shown in this thesis were promising enough, a reduction in total wire-length and area has been achieved, while at the same time physical design process has been accelerated. However, a drawback of the proposed methodology the lack of automation because of the strong association between the designer and each step of each presented flow. The RTL implementation has a strong impact at the final results. Therefore, this association is crucial, making the proposed methodologies sometimes inefficient because of the required designer-procedure interaction. The reduction of this necessary interaction is part of the future work discussed in the following section.

7.2 Future Work

One way to improve the efficiency of the proposed methodologies is by introducing more automation into the flow. This can be achieved by reducing the essential interaction between the designer and the procedure. One approach to increase automation is to formulate the problem as a linear equation $Ax = b$, which can be solved using algorithms that require less human intervention. By introducing cell properties in the flow, more specifically, in the table A formulation, may help find the best solution and, at the same time, reduce the amount of time spent on manual tasks.

In recent years, many authors have stated that the use of Artificial intelligence (AI) is becoming increasingly important in the design process. Exploiting the power of AI and Machine Learning could help automate tasks and improve the efficiency of the design flow. For example, Verilog analysis and regularity extraction, as well as the placement of cell groups can be accomplished with the help of introduced intelligence in the flow. Additionally, AI can be used to analyze and optimize the performance of a design, leading to better results and a more efficient design process.

Another approach to improve the design process is to modify placement and floorplan steps to be top level aware. By taking a holistic approach to the design, designers can ensure that each element of the design is optimized for performance and efficiency. By considering the top-level layout during placement and floorplan steps, designers can avoid potential problems that may arise, from the hierarchical approach, later in the design process. This approach can help to minimize the number of design iterations needed, saving time and improving the overall quality of the design.

Bibliography

[1] C.-C. Huang, B.-Q. Lin, H.-Y. Lee, Y.-W. Chang, K.-S. Wu, and J.-Z. Yang, "Graph-based logic bit slicing for datapath-aware placement," in *Proceedings of the 54th Annual Design Automation Conference 2017*, ser. DAC '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3061639.3062254

[2] C. P. Sotiriou, N. Sketopoulos, A. Nayak, and P. I. Pénzes, "Extraction of structural regularity for random logic netlists," *2019 Panhellenic Conference on Electronics & Telecommunications (PACET)*, pp. 1–7, 2019.

[3] H. Xiang, M. Cho, H. Ren, M. Ziegler, and R. Puri, "Network flow based datapath bit slicing," in *Proceedings of the 2013 ACM International Symposium on Physical Design*, ser. ISPD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 139–146. [Online]. Available: https://doi.org/10.1145/2451916.2451954

[4] A. Chowdhary, S. Kale, P. Saripella, N. Sehgal, and R. Gupta, "Extraction of functional regularity in datapath circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 9, pp. 1279–1296, 1999.

[5] Arikati and Varadarajan, "A signature based approach to regularity extraction," in *1997 Proceedings of IEEE International Conference on Computer Aided Design (ICCAD)*, 1997, pp. 542–545.

[6] "GENUS (Synthesis Solution) Tool."

[7] "ASP (Automated Structured Placement) Tool."

[8] "TCL (Tool Command) Language."