



## Article

# Methodology for Structured Data-Path Implementation in VLSI Physical Design: A Case Study

Dhilleswararao Pudi <sup>1</sup>, Samuel Jigme Harrison <sup>2</sup>, Dimitrios Stathis <sup>3</sup>, Srinivas Boppu <sup>1</sup>, Ahmed Hemani <sup>3</sup>  
and Linga Reddy Cenkeramaddi <sup>4,\*</sup>

<sup>1</sup> School of Electrical Sciences, Indian Institute of Technology Bhubhaneswar (IITBBS), Bhubaneswar 752050, Odisha, India

<sup>2</sup> ST Microelectronics Asia Pte. Ltd., Singapore 554574, Singapore

<sup>3</sup> School of Electrical Engineering and Computer Science, Royal Institute of Technology (KTH), Stockholm 16640, Sweden

<sup>4</sup> Department of Information and Communication Technology, University of Agder, 4879 Grimstad, Norway

\* Correspondence: [linga.cenkeramaddi@uia.no](mailto:linga.cenkeramaddi@uia.no)

**Abstract:** State-of-the-art modern microprocessor and domain-specific accelerator designs are dominated by data-paths composed of regular structures, also known as bit-slices. Random logic placement and routing techniques may not result in an optimal layout for these data-path-dominated designs. As a result, implementation tools such as Cadence's Innovus include a Structured Data-Path (SDP) feature that allows data-path placement to be completely customized by constraining the placement engine. A relative placement file is used to provide these constraints to the tool. However, the tool neither extracts nor automatically places the regular data-path structures. In other words, the relative placement file is not automatically generated. In this paper, we propose a semi-automated method for extracting bit-slices from the Innovus SDP flow. It has been demonstrated that the proposed method results in 17% less density or use for a pixel buffer design. At the same time, the other performance metrics are unchanged when compared to the traditional place and route flow.

**Keywords:** data-path; placement; routing; innovus; electronic design automation; physical design



**Citation:** Pudi, D.; Harrison, S.J.; Stathis, D.; Boppu, S.; Hemani, A.; Cenkeramaddi, L.R. Methodology for Structured Data-Path Implementation in VLSI Physical Design: A Case Study. *Electronics* **2022**, *11*, 2965. <https://doi.org/10.3390/electronics11182965>

Academic Editors: Fábio Passos, Nuno Lourenço and Ricardo Martins

Received: 10 August 2022

Accepted: 15 September 2022

Published: 19 September 2022

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

In the era of the deep-submicron, System-on-Chip (SOC) design has become a daunting and challenging task. Aggressive performance goals and stringent time-to-market deadlines put huge pressure on design teams to deliver the masks in time for production, validate the design, build a prototype, and make the chip available commercially. The Electronic Design Automation (EDA) industry is the key enabler for the growth and huge success of the entire VLSI design community by reducing the gap between design complexity and productivity. Computer-Aided Design (CAD) tools automate many tasks, such as verification, synthesis, timing analysis, and physical design. In particular, physical design and implementation tools help in achieving higher quality (in terms of performance goals such as area, timing, and power) with shorter execution times. However, the increased complexity of modern SoCs, growing design sizes, and the introduction of new technologies pose challenges to the CAD tools developed by the EDA industry. Standard cell placement is a critical step in physical design, which determines the quality of the layout. The goal of any placement tool or placer is to meet the timing constraints while minimizing the area, power, and wire length. The placement engines or algorithms run iteratively to achieve the best Quality of Results (QoR). Innovus from Cadence is one of the physical design tools which automates the whole physical design flow, including standard cell placement.

Data-paths are common in modern microprocessors, graphics processors, and domain-specific accelerators, where the same logic is repeated multiple times. The structure of a data-path is highly regular, and it typically includes registers, multiplexers, and arithmetic

circuits such as adders and multipliers. To produce the output, each bit in a data-path goes through the same data operations. The data-path is the most important and crucial component of the entire design. As a result, designing the data-path and its layout is critical for the chip's overall performance goals. Data-path designs are placed in a bit-sliced pattern to achieve the best results, such as area, timing, and power. Traditionally, these data-path elements can be hardened and brought as macros in higher-level designs. However, this approach is not comfortable since all the physical and timing views need to be generated whenever there is a change in the data-path—leading to multiple iterations and increased turnaround time. It would be preferable to work with high-level modules that incorporate both random and data-path logic and do the necessary data-path customization to achieve the best results. The placer of the Innovus does achieve good QoR; however, if the design has Structured Data-Path (SDP) elements, the tool does not identify them and therefore places them randomly. This might result in a compromise of QoR. Innovus can handle the SDP elements using a relative placement file; however, by default, it will not identify them. The tool vendors provide some initial scripts and recommendations to extract SDP structures; since there are no deterministic algorithms it cannot be fully automated. However, if the implementation engineer have a good idea of the design, the tools will help us in extracting SDP structures. The main contributions of the proposed work are:

- We evaluate the SDP flow of Cadence's Innovus using two case studies, summarize our design experience, and discuss the results.
- A generic methodology is proposed to use the SDP flow of Innovus to achieve the best Quality of Results (QOR).
- The proposed methodology recommends the extraction of bit-slices of the design in a semi-automated way to generate a placement file, which orients the instances in the design such that routing between the bit-slices is minimized, and places the instances of the SDP with required gaps to reduce the vertical or horizontal congestion.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 gives brief details of the designs that are considered for evaluating the methodology. Section 4 describes our methodology. Experiments and results of our methodology are discussed in Sections 5 and 6, respectively. Concluding remarks are given in Section 7.

## 2. Related Work

The structured data-paths are regular in their structure, and their careful placement is essential for obtaining a good quality layout. Chou et al. in [1] present an algorithm that can exploit the regular structures in the design, i.e., it can extract them and place them appropriately to minimize the wire length, etc. In [2], Ye et al. talk about capturing data-path bit-sliced structures using Abstract Physical Model (APM), which can be extracted from the data flow graph. This APM can be further used for interconnect and congestion planning. This methodology was implemented in C++ and benchmark data-path circuits, which is an FIR filter. In [3], Nijssen et al. present an algorithm for the automatic extraction of regular structures from the logic netlist of data-paths for better placement. In [4], Ward et al. discuss extracting data-paths using a novel data learning technique and a new placement algorithm. In [5], Serdar et al. present a design flow for automatic data-path placement using simulated annealing and a global placement flow based on the o-tree algorithm. Liew et al. in [6] proposed a manual placement algorithm for handling SDP elements using Integrated Circuit Compiler I (ICC I) from Synopsys. In their approach, from the initial ICC placement, structured registers, including their connections, are extracted to form RP (Relative Placement) groups. Later, these extracted RP groups were placed manually using an algorithm named SDP-RP using ICC I. Synopsys also introduced ICC II, which has an SDP flow similar to Cadence's Innovus SDP flow. In [7], Henrik et al. used ICC II SDP flow for a synthesizable register file design. He also extracted the SDP structures from the design using script named *pyplace* and generated the relative placement file as per the syntax of ICC II. However, the results of the SDP flow of ICC II were not as good as the normal flow of ICC II, where the tool does everything, i.e., no manual intervention.

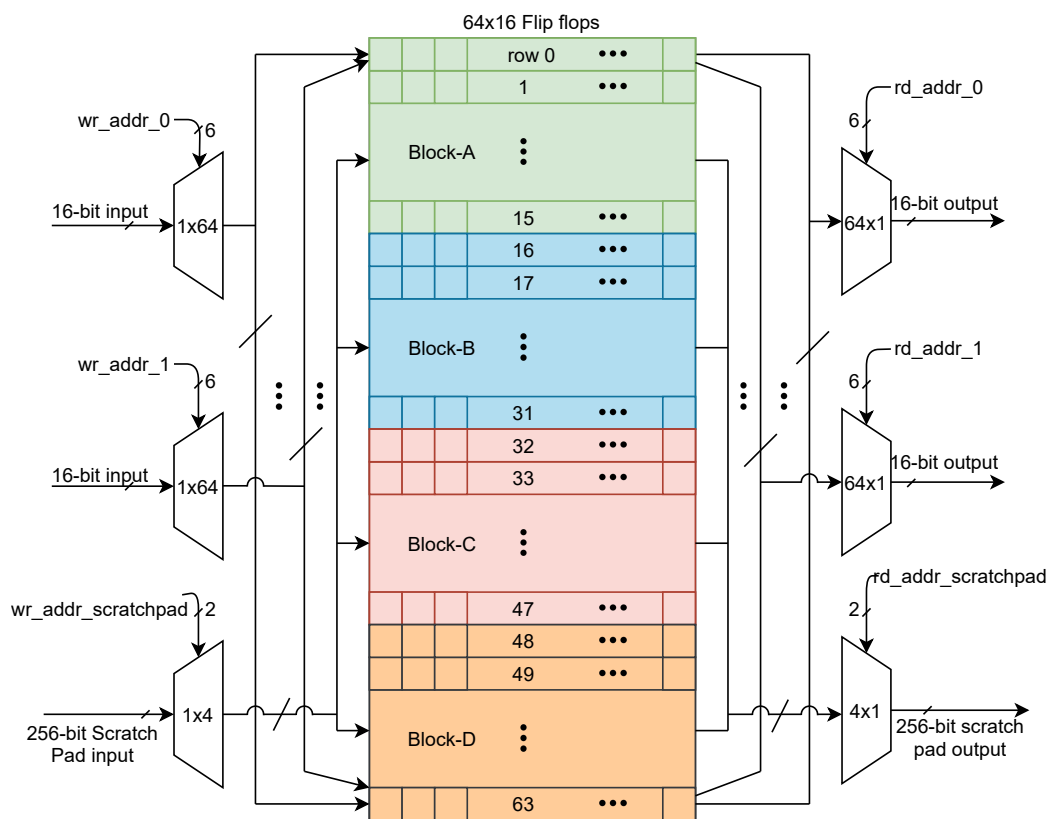
We presume that the relative placement file did not capture all the SDP elements leading to poor results. Sabyasachi et al. in [8] presented a methodology for detailed routing of data-path connections that are regular in nature. Similarly, structured nets across various bit-slices are identified as net clusters. For each representative net of the cluster, an optimal route was identified first, and the same routing was inferred for the rest of the nets. This approach results in more regular connections, and since only a few nets are routed, it will also result in significant speedup. In [9], Sotiriou et al. presented two algorithms named Greedy and Isomorphism to extract SDP clusters from synthesized gate-level netlists. In the same paper, they also presented an SDP placement algorithm—compatible with commercial tools—to place the extracted SDP clusters. Currently, the most widely used tools ICC and Innovus from Synopsys and Cadence, respectively, support the SDP flow. However, these tools do not extract the SDP elements automatically and place them. Though there are many algorithms proposed to extract data-path elements automatically in [1–6], design knowledge is essential to extract the data-path structures and place them; therefore, none of the commercial tools have automatic data-path extraction in their flow. Therefore, extracting the SDP elements is an essential step in custom placement of them. Our work in this paper extracts the SDP elements in the design using scripts based on the design knowledge and the structure of the generated netlist. Furthermore, we evaluate the SDP flow of Innovus and present a methodology for achieving optimal results. To the best of our knowledge, this is the first work to try and evaluate the SDP flow of Innovus.

### 3. Designs for Evaluation

In this section, a brief overview of two designs that are considered for the evaluation of SDP flow is presented. The first design is a *Register File* (RFile) of a Coarse Grain Reconfigurable Array (CGRA), and the second design is a *Pixel Buffer* (PB) of an Adaptive Median Filtering (AMF). These minimal design details will help us understand the methodology and the results sections.

#### 3.1. Register File

A CGRA [10] fabric consists of an array of Processing Elements (PEs) arranged in two rows and an arbitrary number of columns. The Register File (RFile) is an important component of the PE that provides local storage to the computational units. Raw data from a scratchpad memory is written to an RFile for processing, and processed data are read back again into the scratchpad memory. For further details of the CGRA design, we refer to [11,12] and restrict our discussion in this paper to RFile only. The design of the RFile is shown in Figure 1, where a register row is instantiated 64 times, and each row is of 16-bit length—it can be easily visualized as 16 D flip-flops in a row. The RFile has two read and write ports, each with a dedicated address that points to one row of the RFile for read/write. This is shown in Figure 1 using  $1 \times 64$  multiplexer or  $64 \times 1$  multiplexers. The RFile also has one 256-bit bidirectional port, which is dedicated to the data transfer between scratchpad memory and the RFile. The data in one row of the scratchpad memory occupies 16 successive locations of RFile and vice versa, as shown in Figure 1 using Block-A, B, C, or D. To select one of these blocks for either read/write operation, a dedicated read/write address is provided using multiplexers associated with the scratchpad interface, see Figure 1.



**Figure 1.** Register File Architecture

### 3.2. Pixel Buffer

In image processing, median filters are often used to remove the impulse noise [13]. However, if the probability of impulse noise becomes high, a standard median filter with a window size of either  $3 \times 3$ ,  $5 \times 5$ ,  $7 \times 7$ , or  $9 \times 9$  is used depending on the percentage (%) of noise [14]. Since the amount of noise is random and unknown a priori in adaptive median filters, the window size is chosen dynamically or at run-time [15]. For further details of the adaptive median filters, we refer to [13,15]. In such adaptive filters, pixel buffers are often needed in the hardware, cf. Figure 2. In this pixel buffer of  $9 \times 9$ —arranged in a 2-D matrix style—each element is an 8-bit D-type register (flip-flop) storing the neighborhood of all the window sizes. The filtering algorithm can adaptively choose a window size, calculate the median and determine whether a center pixel is noisy or not, and replace it if needed. Pixel buffer supports raster scanning of the image where 9 pixels per clock can enter it either from right  $Z_x$  or left  $X_x$ , controlled by a multiplexer with a control signal, *right\_left\_ctrl*, see Figure 2. To maximize the reuse of the pixels present in the buffer, at both column ends of an image, the pixel values are shifted in a bottom-up fashion. Here, the new pixels enter from the bottom ( $Y_x$ ), controlled by a multiplexer with a control signal (*bottom\_ctrl*). To summarize, the pixel buffer consists of 8-bit shift registers in a  $9 \times 9$  matrix form, supporting right, left, and bottom-up data movement.

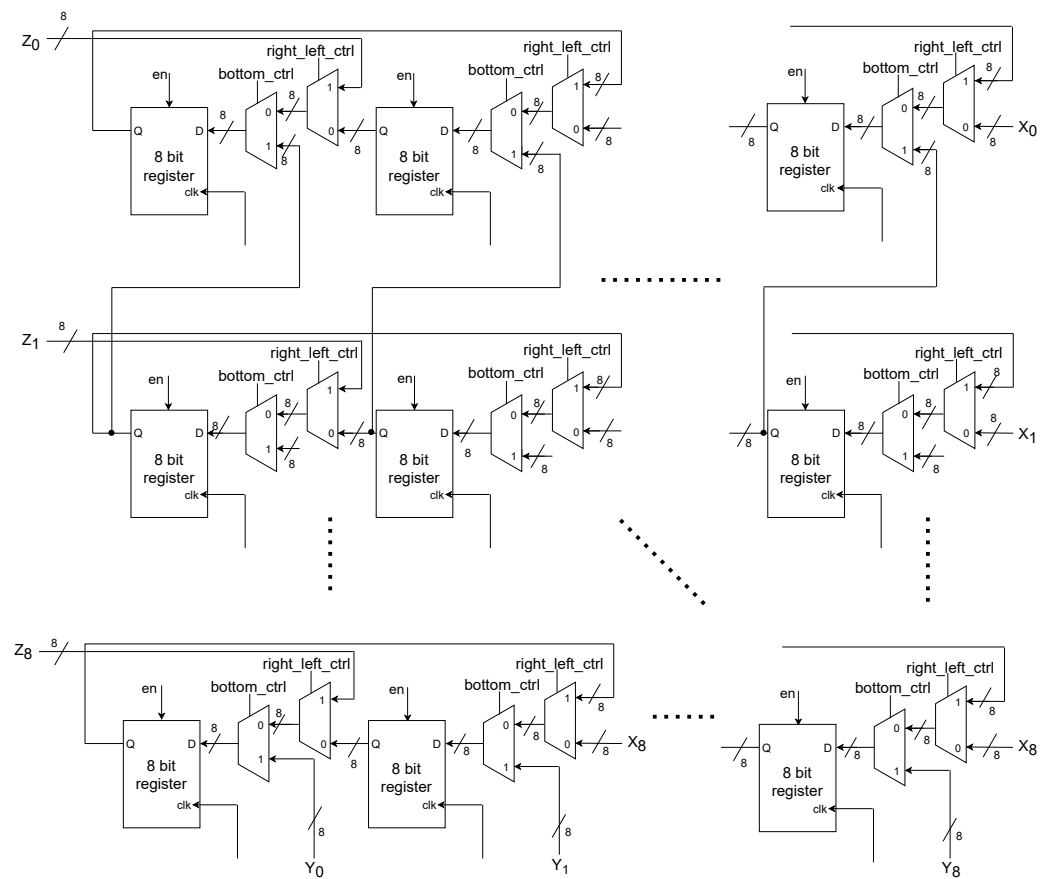


Figure 2. Pixel Buffer Architecture

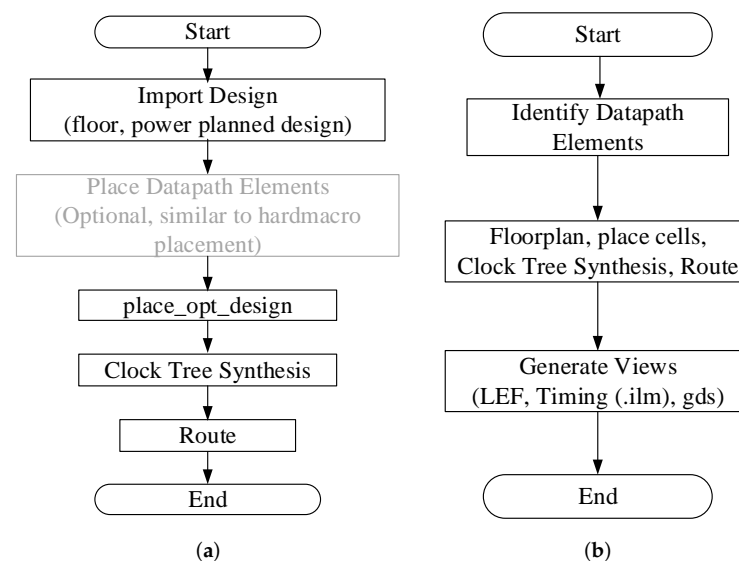
#### 4. Methodology

In this paper, our focus is on the data-path of a design that is already synthesized using standard cells of a given technology. The designer extracts data-path bit-sliced structures manually or semi-automatically for a given design. Such a bit-sliced structure benefits are straight and short control and data signal routes. By placing the data-path in a bit-sliced pattern, the area is reduced because of the abutted cells, and the signal and clock skews are minimized. The aligned cells create straight wires reducing vias and jogs, resulting in better timing, power, and congestion. However, the design and implementation are often done by different people; thus, identifying the bit-sliced patterns might not be easy by just looking at the synthesized netlist.

##### 4.1. Traditional Design Flow

In a traditional or normal standard cell-based design flow, the implementation steps shown in Figure 3a are followed, which includes: (a) importing a floor-planned, power-planned design, (b) placing the standard cells, (c) synthesizing the clock tree, and (d) routing the design. Optimizations can be performed at each step to improve the performance metrics such as area, timing, and power. The same standard flow can be followed for the data-path-based designs where the tool places the data-path standard cells along with other (logic and control) standard cells. In this flow, the advantage is that no manual intervention is needed, the flow is standardized, and most importantly, detailed design knowledge is not required. Albeit the tool might do the job, but there is still a possibility to improve the timing and reduce the use and power. To overcome this challenge, a data-path hardening flow can be used as shown in Figure 3b. In this flow, one needs to identify the bit-sliced patterns of the data-path and harden them using the standard flow shown in Figure 3b. These hardened data-path structures are similar to hard macros, which are brought into the traditional design flow, see Figure 3a. These macros can now be placed per the design

requirements, which might improve the performance metrics. Though this approach is promising, it has many disadvantages and is error-prone. For instance, for all the data-path structures, physical views such as LEF, GDS, etc., and timing views such as ILM, ETM, etc., need to be generated, which is time-consuming. Furthermore, these views need to be generated whenever there is a change to the datapath structure, or the hardening of the datapath structure does not yield good results. In this case, re-hardening of the datapath structure is needed to improve the results further. To overcome these challenges, the SDP flow of Innovus is highly recommended. The datapath structures in the SDP flow of Innovus are handled similar to hard macros by which the placement can be customized using the available standard cell views. There is no need for generating either physical or timing views.



**Figure 3.** Traditional Design flow for Standard cell-based Design. (a) Traditional Design Flow. (b) Datapath Hardening Flow.

#### 4.2. SDP Flow

The SDP flow is recommended for structured data-paths where better performance, power, and area are required. Since it is a semi-custom methodology, one must have a detailed understanding of the design to improve these performance metrics. Currently, Innovus does not identify bit-sliced patterns automatically. Based on the design knowledge, a script must be written to extract the bit-sliced patterns of the data-path for placement. This flow might not yield good results if these patterns are not extracted and appropriately placed. The main benefits of SDP flow are controlled placement throughout the flow, which might result in uniform routing, and improved performance metrics.

Figure 4 shows the SDP flow for standard cell-based data-path dominating designs. Here, the placement of the bit-sliced structure is provided to the tool using a *Relative Placement File* (RPF) or TCL commands. However, it is highly recommended to use the RPF and generate it using either Perl or Python scripts to try out different placement scenarios. For the syntax of this file, rules, and notations, we refer to [16]. An exemplary RPF is shown in Figure 5a, and the corresponding placement is shown in Figure 5b.

In RPF notation, the entire core area is divided into either rows or columns, where a row can have multiple columns, or a column can contain multiple rows. In RPF, nesting of rows inside another row or nesting of columns inside another column is not allowed. However, nesting of rows inside a column or columns inside a row is allowed with even deep nesting. In Figure 5, the entire core area is visualized as a single row (R1), though physically, it has four rows, see Figure 5b. In R1, using column C1, instance U5\_0 is placed first, followed by skipping of two rows (using *skipspace* 2), and instance U5\_1 is finally placed. The *origin* statement controls the start of the row, and the *justifyBy* statement



controls the justification or alignment of the row. In the same R1, using column C2, instances U5\_2, U5\_3, U5\_4, and U5\_5 are placed vertically. To have some empty space between columns C1 and C2, *skipspace 10* was used, which leaves 10 M2 tracks between C1 and C2. In this way, a very detailed and fine placement of data-path structures can be done as per the user’s requirements. There are more constructs that can be used in RPF, which allow us to control the orientation (MX, R180, MY, etc.) of instances, justify rows and columns (SW, NW, etc.), align instances by pin name, the origin of a row or column, etc. [16].

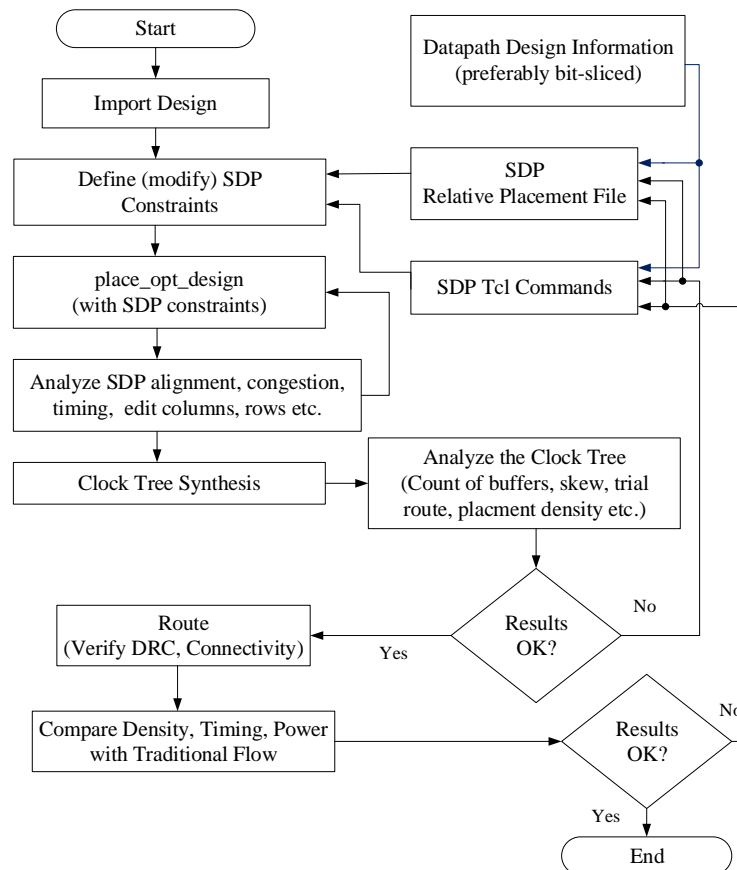


Figure 4. SDP-based Standard Cell Design Flow.

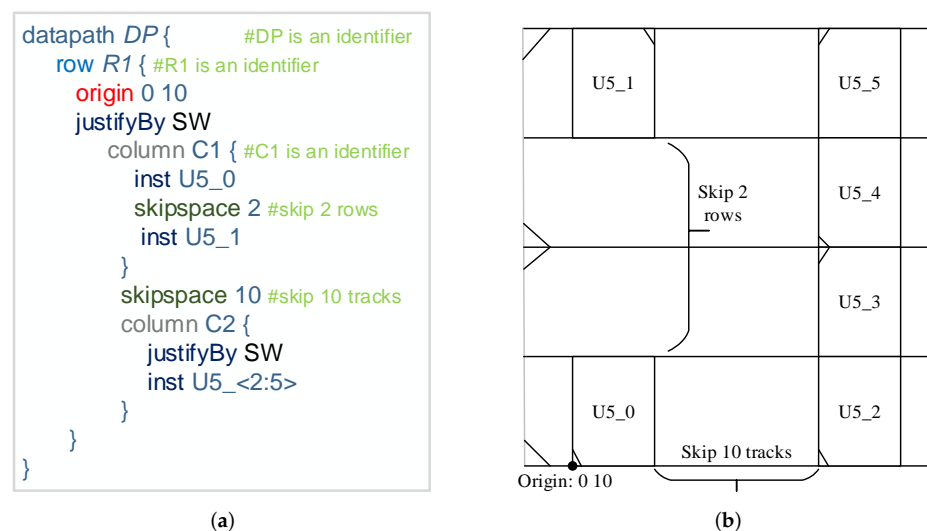
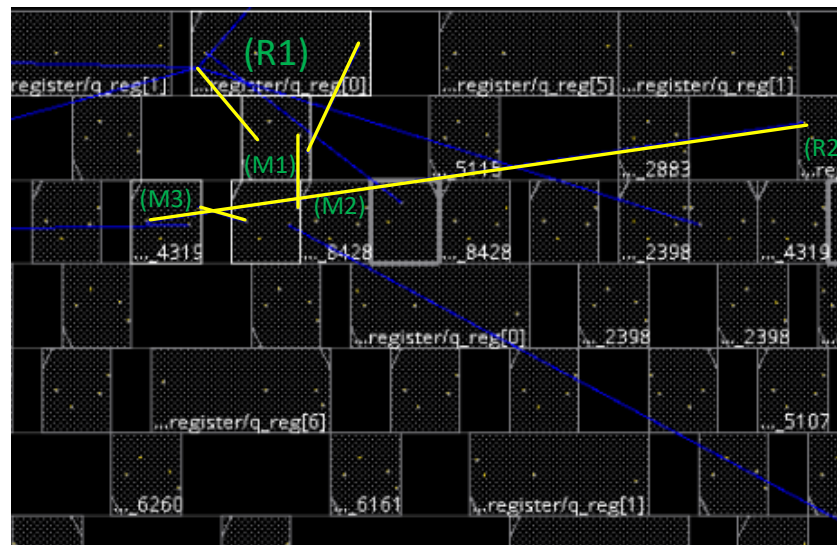
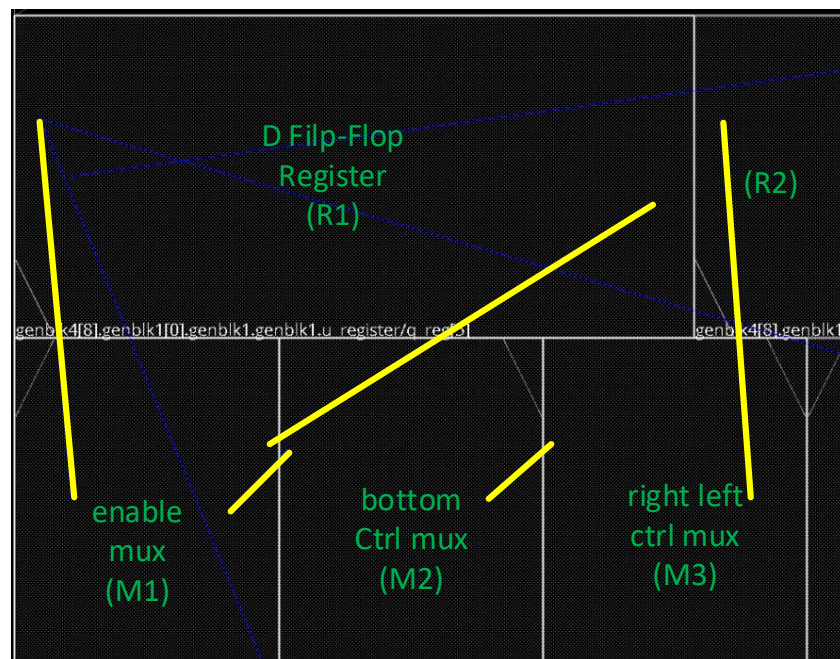


Figure 5. Relative Placement File and its Corresponding Placement. (a) Relative Placement File; (b) Placement of Cells.

Figure 6 shows the placement of a D-Flip-Flop and associated multiplexers for the pixel buffer design. A D Flip-Flop has an enable pin; therefore, it results in one more additional multiplexer (M1) apart from the right\_left\_control multiplexer (M3) and bottom\_control multiplexers (M2), see Figures 2 and 6. Placement of the multiplexers and D-FF is shown in Figure 6a for normal flow and in Figure 6b for SDP flow. All the connections among these cells are highlighted in yellow in both cases. In the SDP flow, since the placement is controlled using the RPF, cells are placed very close to each other, whereas, in the normal flow, cells are placed apart, see Figure 6. Therefore, generating the right RPF file is important in the SDP flow.



(a)



(b)

**Figure 6.** Placement of D-FF and Multiplexers in both Normal and SDP Flow. (a) Placement in Normal Flow. (b) Placement in SDP Flow.



Once the RPF file is read into Innovus, placement of these SDP structures can be visualized in GUI and, if needed, RPF can be modified. Once the placement of SDP elements is finalized, the standard steps shown in Figure 4 can be followed, which are more or less similar to the traditional flow. However, after each step in the flow (i.e., placement, clock tree synthesis, or routing), use, timing, power, etc., can be compared to the normal flow. If only the results are satisfied, we proceed to the next flow step. Else, we analyze the design and modify the RPF before restarting the flow, see Figure 4. Multiple iterations are required to arrive at the best RPF that improves the performance metrics. The SDP flow of Innovus makes it easy to perform these iterations.

## 5. Experiments

To evaluate the SDP flow of Innovus, both the RFile and Pixel Buffer (PB) designs were considered, and the implementation was made with and without SDP flow as per the steps shown in Figures 3a and 4, respectively. The RPF files for both designs were generated using Python scripts. Based on the design knowledge, the bit-sliced data-path structures were extracted for both designs. For the RFile design, an RPF file was generated with the following design information. In the RFile design, each row contains 16 D-FFs, and each FF is independent (unlike shift registers, no data transfer within rows); therefore, these FFs were placed in the same row. Furthermore, to allow routing to these registers, sufficient space was left between them. The same bit registers in each row were also aligned to enforce the straight routes since the same input might enter one of these registers. This information was easy to extract from the design, and an RPF file was generated with this information. However, the RFile design contains a lot of logic (multiplexers/demultiplexers with read/write addresses, see Figure 1) associated with the read/write ports of RFile itself or the scratchpad interface. However, extracting the bit-sliced patterns from this logic is tough; since one must understand the synthesized netlist and identify the structured patterns. Therefore, the generated RPF does not contain any information about the placement of this mux/demux logic. This logic is placed using the tool itself after the placement of the SDP structures.

For the PB design, extracting the data-path structure was relatively easy. The PB buffer is in the form of a  $9 \times 9$  matrix where each element is an 8-bit FF. All the elements have a shift register structure, i.e., 8-bit data can be pushed from left, right, or bottom. Since each element is an 8-bit register, it can be thought of as a  $9 \times 9$  structure shown in Figure 2 existing for each bit—in total, eight structures. Furthermore, all these eight structures are independent, i.e., there are no connections among these structures. All the connections exist within the structure itself, which also means that these structures can be placed independently. An RPF was generated to capture the above information and each structure was placed together and away from the others. Once the RPF was generated for the designs, the flow steps shown in Figure 4 were followed to complete the implementation.

## 6. Results and Discussion

The results for both RFile and PB designs are summarized in Tables 1 and 2, respectively. Figure 7a shows the layout of the RFile design using normal design flow, whereas Figure 7b shows the layout of the RFile design with the placement of its rows using the RPF. It can be observed from Figure 7a,b that placement of registers is much more regular and uniform in SDP flow, resulting in low placement density compared to normal flow. In Table 1, the worst negative slack (WNS), total negative slack (TNS), density, power, congestion, total net length, and clock cell area were compared for both normal and SDP flow for the same core area. It can be observed that the SDP flow results were not as good as the normal flow since the RPF contains only the rows of the register file, as shown in Figure 7b. Due to the lack of design knowledge, no more information was added to the RPF, which resulted in poor results. The rest of the mux/demux logic in the design was placed all over the chip, highlighted by M2 routing alone (red color); see Figure 7b. Total net length, number of vias, and wire capacitance were high, resulting in more power consumption and

less TNS though the density was a bit better in the SDP flow than in normal flow. Since the FFs were placed regularly, clock cell count or area was less in SDP compared to normal flow. In the normal flow of RFile design, Innovus was able to optimize the placement of not only the register rows but also mux/demux logic, see Figure 7a, resulting in better performance metrics. To summarize, the SDP flow did not yield better results, proving that it is essential to extract the bit-sliced patterns of data-path design for the success of the SDP flow.

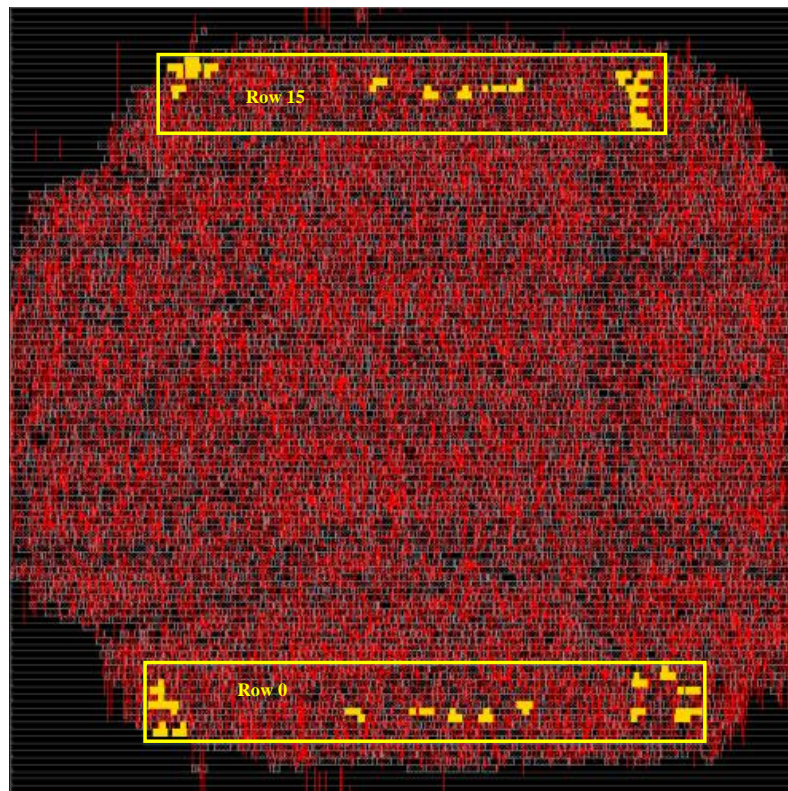
Figure 8b shows the layout of the PB design with the placement of data-path structures using the RPF. As aforementioned, a similar structure exists for each bit of the registers in the design, as shown in Figure 2. Each such structure can be placed regularly, as shown in Figure 8b; the numbers in the yellow boxes denote the bit positions. Figure 8a shows the layout of the PB design using normal flow. Table 2 compares various performance metrics for normal and the SDP flow using the same core area. For the PB design, the SDP flow results were better since the RPF file was complete, i.e., all the needed bit-sliced information was captured. It can be observed that SDP flow has 17% less density compared with the normal flow; see the empty space in Figure 8b. In normal flow, the instances were placed all over the core area, see Figure 8a. Figure 9a,b show the clock tree network for the normal and SDP flow, respectively. In both cases, the clock has straight routes; however, in the SDP flow, the clock net length or total wire length was less, resulting in slightly lower power consumption. Timing in the SDP flow was slightly degraded; however, the use/density and other metrics were lower for the SDP flow. SDP flow is present in both Cadence's Innovus and Synopsys' ICC II; however, it is still not a part of the standard design flow because its quality of results depends on effective data-path extraction, for which there are no deterministic algorithms. Therefore, to avail of the benefits of the SDP flow, design knowledge is essential to extract the data-path components for optimal placement using an RPF.

**Table 1.** Summary of Results for RFile Design, Core Area  $W = 179.6 \mu\text{m}$ ,  $L = 179.55 \mu\text{m}$ .

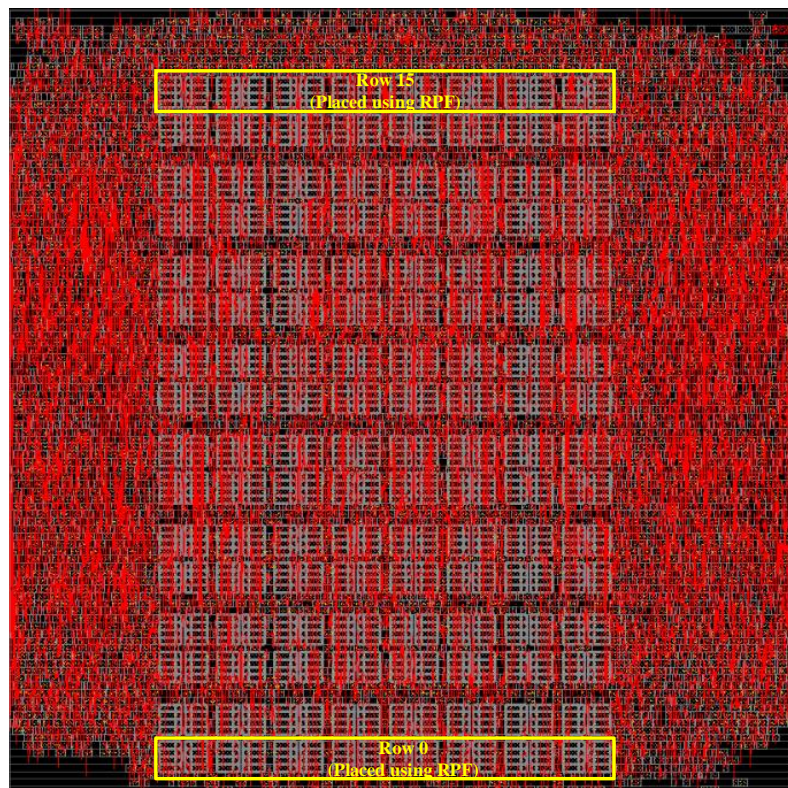
Flow Step	WNS (ns)		TNS (ns)		Density (%)		Power (mW)		Congestion (%)		Total Net Length ( $\mu\text{m}$ )		Clock Cell Area ( $\mu\text{m}^2$ )		#Vias	
	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP
Placement	9.270	8.622	0	0	73.599	63.747	-	-	0	0% <sub>H</sub> , 1.54% <sub>V</sub>	-	-	-	-	-	-
CTS	9.251	8.605	0	0	74.068	71.853	-	-	0	0% <sub>H</sub> , 1.81% <sub>V</sub>	-	-	-	-	-	-
Route	9.229	7.354	0	0	74.068	71.853	0.777	0.91	0	0% <sub>H</sub> , 1.92% <sub>V</sub>	$1.34 \times 10^5$	$2.34 \times 10^5$	296.514	272.232	65,822	67,867

**Table 2.** Summary of Results for PB Design, Core Area  $W = 114.8 \mu\text{m}$ ,  $L = 107.73 \mu\text{m}$ .

Flow Step	WNS (ns)		TNS (ns)		Density (%)		Power (mW)		Congestion (%)		Total Net Length ( $\mu\text{m}$ )		Clock Cell Area ( $\mu\text{m}^2$ )		#Vias	
	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP	Normal	SDP
Placement	9.077	8.971	0	0	69.886	0	-	-	0	0	-	-	-	-	-	-
CTS	9.055	8.961	0	0	70.58	52.824	-	-	0	0	-	-	-	-	-	-
Route	9.043	8.968	0	0	70.58	52.824	0.4516	0.4427	0	0	$1.61 \times 10^4$	$1.45 \times 10^4$	85.842	85.5	17,802	15,423



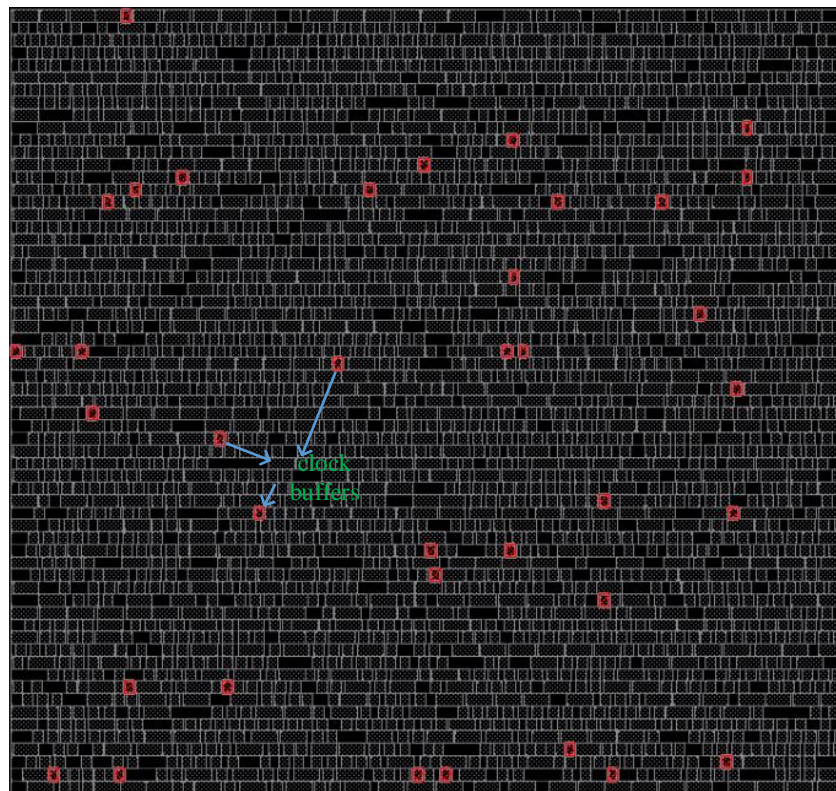
(a)



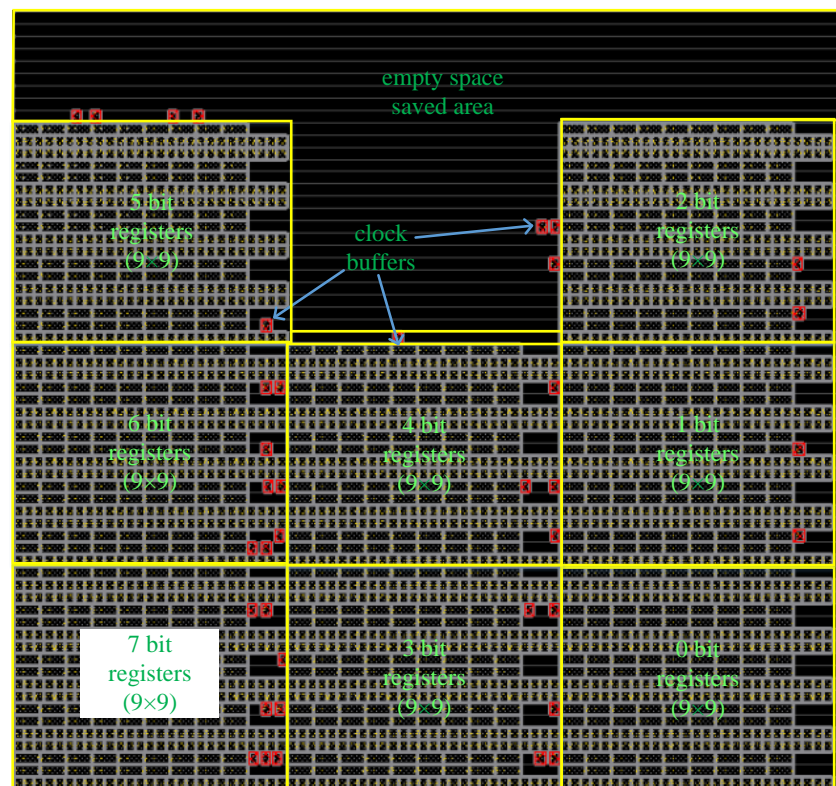
(b)

**Figure 7.** Layout of RFile in both Normal and SDP Flow. (a) Layout of RFile in the Normal Flow. (b) Layout of RFile in the SDP Flow.



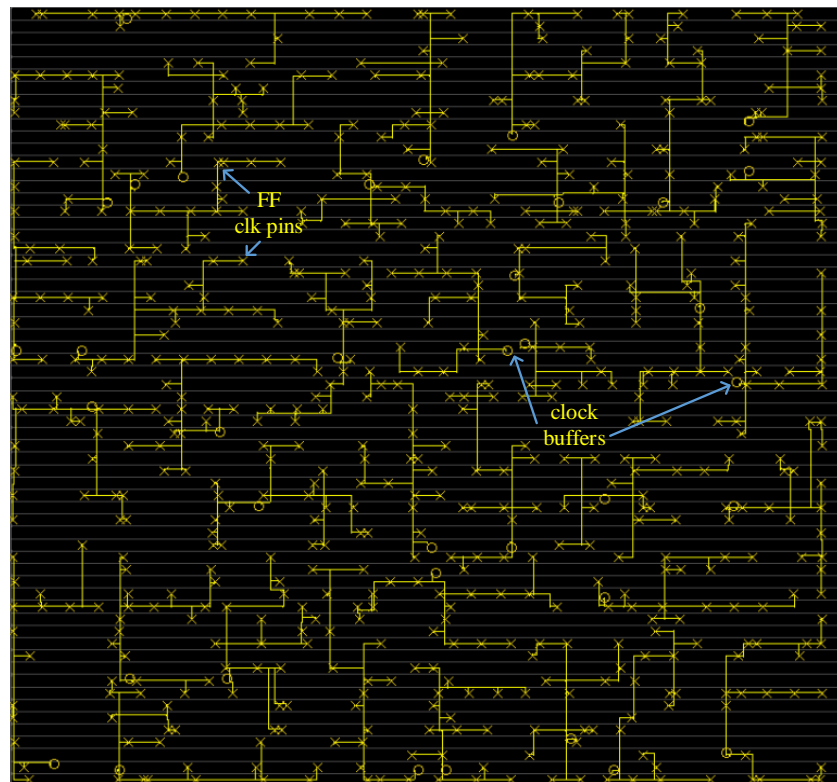


(a)

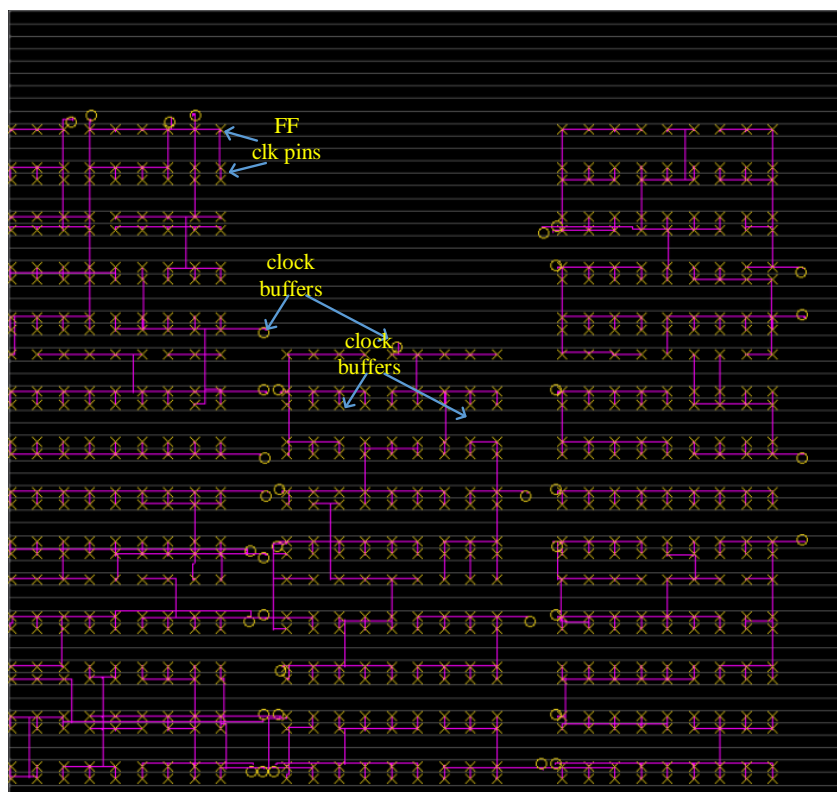


(b)

Figure 8. Layout of Pixel Buffer in both Normal and SDP Flow. (a) Normal Flow. (b) SDP Flow.



(a)



(b)

**Figure 9.** Clock Tree in both Normal and SDP Flow. (a) Normal Flow. (b) SDP Flow.



## 7. Conclusions and Future Work

This paper evaluated the Structured Datapath Placement (SDP) flow of Cadence's Innovus using a register file and pixel buffer designs. The bit-sliced structures of the designs were extracted based on the design knowledge and synthesized gate-level netlist. A relative placement file was generated using Python scripts where the bit-sliced information was represented using the supported statements or syntax of the tool. For both designs, the SDP flow and traditional or normal flow results were compared in terms of utilization or density, timing, power, congestion, clock cells' area, total wire length, and the number of vias in the design. SDP flow did not show any improvement for the register file design compared to the normal place and route flow. The register file design contains register rows and multiplexer and demultiplexer logic controlling the reads/writes from/to it. For the multiplexer and demultiplexer logic, bit-sliced structures could not be identified using the gate-level netlist. Therefore, placement constraints were not captured in the relative placement file resulting in poor performance metrics. However, for the pixel buffer design, the SDP flow shows significant improvements over the normal flow. It shows 17% less use while the rest of the performance metrics almost remains the same. The relative placement file captured all the structural details and placement constraints. Based on our evaluation, the SDP flow is recommended for structured data-path placement; however, the success of the flow depends on the finer or detailed extraction of the data-path structures and generating placement constraints, i.e., relative placement file. In the future, we would like to work on coding guidelines to extract data-path easily by the tools and a more structured and automated approach for generating the relative placement file.

**Author Contributions:** Conceptualization, D.P. and S.B.; Methodology D.P. and S.B.; Technical and Tool support, D.P., S.B. and S.J.H.; Validation D.P.; resources D.P., S.B. and S.J.H.; formal analysis D.P.; investigation, D.P.; data curation, D.P.; writing—original draft, D.P., and S.B.; writing—review and editing, D.P., S.B., D.S., A.H. and L.R.C.; project administration, A.H.; Funding acquisition, L.R.C. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was partly supported by the Indo-Norwegian Collaboration in Autonomous Cyber-Physical Systems (INCAPS) project: 287918 of the International Partnerships for Excellent Education, Research and Innovation (INTPART) program from the Research Council of Norway.

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Chou, S.; Hsu, M.K.; Chang, Y.W. Structure-aware placement for datapath-intensive circuit designs. In Proceedings of the 49th Annual Design Automation Conference, San Francisco, CA, USA, 3–7 June 2012; pp. 762–767.
2. Ye, T.T.; De Micheli, G. Data path placement with regularity. In Proceedings of the IEEE/ACM International Conference on Computer Aided Design, San Jose, CA, USA, 5–9 November 2000; pp. 264–270.
3. Nijssen, R.X.; Jess, J.A. Two-dimensional datapath regularity extraction. In *IFIP Workshop on Logic and Architecture Synthesis*; Citeseer: Princeton, NJ, USA, 1996; pp. 110–117.
4. Ward, S.; Ding, D.; Pan, D.Z. PADE: A high-performance placer with automatic datapath extraction and evaluation through high-dimensional data learning. In Proceedings of the DAC Design Automation Conference 2012, San Francisco, CA, USA, 3–7 June 2012; pp. 756–761.
5. Serdar, T.; Sechen, C. Automatic datapath tile placement and routing. In Proceedings of the Design, Automation and Test in Europe. Conference and Exhibition 2001, Munich, Germany, 13–16 March 2001; pp. 552–559.
6. Mei, L.Y.; Rosdi, B.A.B.; Kok, L.C. A methodology for automation structured datapath placement In VLSI design. In Proceedings of the 2011 IEEE Symposium on Industrial Electronics and Applications, Langkawi, Malaysia, 25–28 September 2011; pp. 273–278.
7. Sørvik, H.S. Register File Optimization. Master's Thesis, Norwegian University, Trondheim, Norway, 2017.
8. Das, S.; Khatri, S.P. An efficient and regular routing methodology for datapath designs using net regularity extraction. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2002**, *21*, 93–101. [[CrossRef](#)]
9. Sotiriou, C.P.; Sketopoulos, N.; Nayak, A.; Penzes, P. Extraction of Structural Regularity for Random Logic Netlists. In Proceedings of the 2019 Panhellenic Conference on Electronics & Telecommunications (PACET), Volos, Greece, 8–9 November 2019; pp. 1–7.
10. Shami, M.A. Dynamically Reconfigurable Resource Array. Ph.D. Thesis, KTH Royal Institute of Technology, Stockholm, Sweden, 2012.

11. Farahini, N.; Li, S.; Tajammul, M.A.; Shami, M.A.; Chen, G.; Hemani, A.; Ye, W. 39.9 GOPs/watt multi-mode CGRA accelerator for a multi-standard basestation. In Proceedings of the 2013 IEEE International Symposium on Circuits and Systems (ISCAS), Beijing, China, 19–23 May 2013; pp. 1448–1451. [[CrossRef](#)]
12. Dimitrios, S.; Yu, Y. SiLago Fabric. 2019. Available online: <https://github.com/silagokth/fabric> (accessed on 2 April 2020).
13. Hwang, H.; Haddad, R. Adaptive median filters: New algorithms and results. *IEEE Trans. Image Process.* **1995**, *4*, 499–502. [[CrossRef](#)] [[PubMed](#)]
14. Ahmed, E.S.A.; Elatif, R.E.; Alser, Z.T. Median Filter Performance Based on Different Window Sizes for Salt and Pepper Noise Removal in Gray and RGB Images. *Int. J. Signal Process. Image Process. Pattern Recognit.* **2015**, *8*, 343–352. [[CrossRef](#)]
15. Guo, D.; Qu, X.; Du, X.; Wu, K.; Chen, X. Salt and pepper noise removal with noise detection and a patch-based sparse representation. *Adv. Multimed.* **2014**, *2014*, 682747. [[CrossRef](#)]
16. Cadence Inc. Innovus User Guide. Product Version 20.11. 2020. Available online: <https://www.cadence.com> (accessed on 10 February 2021).