AuxcellGen: A Framework for Autonomous Generation of Analog and Memory Unit Cells

Sumanth Kamineni[1], Arvind Sharma[2], Ramesh Harjani[2], Sachin S. Sapatnekar[2], Benton H. Calhoun[1]
[1]University of Virginia [2]University of Minnesota

***Abstract*—Recent advances in auto-generating analog and mixed-signal (AMS) circuits use standard digital tool flows to compose AMS circuits from a combination of digital standard cells and a set of auxiliary cells (auxcells). Until now, generating auxcell layouts for each new PDK was the last manual step in the flow for auto-generating AMS components, which limited the available auxcells and reduced the optimality of the auto-generated AMS designs. To solve this, we propose AuxcellGen, a framework to auto-generate auxcell layouts and performance models. AuxcellGen generates a parasitic-aware auxcell performance model using a neural network (NN), auto-sizes and optimizes auxcell schematics for a given design target, and auto-generates auxcell layouts. The framework is demonstrated by auto-generating tri-state buffer auxcells for PLLs and sense-amplifier auxcells for SRAM across a range of user specifications that are compatible with standard cell and memory bitcell pitch.**

Keywords: cell-based layout automation, circuit optimization, memory layout generation, surrogate modelling.

I. Introduction

Analog/mixed-signal (AMS) component generators attempt to speed up system-on-chip (SoC) design by replacing the conventional manual design of AMS with automatically generated versions. A fully-autonomous SoC (FASoC) generation framework in [1] uses standard digital tool flows to produce a suite of cell-based AMS generators for components such as low-dropout regulators (LDOs), phase-locked-loops (PLLs), and SRAM memory arrays [2]. The generators in [1] re-cast AMS blocks as structures composed of unit cells, which are either digital standard cells or auxiliary unit cells (“auxcells”) that provide additional functionality unique to the specific AMS block. This permits FASoC [1] to leverage mature digital tool flows for automated layout generation, since the tool flows use structure provided in Verilog to compose layouts from cells by an automatic place and route (APR) process. Fig. 1(a) shows a high-level AMS synthesis flow, which synthesizes the design structure from Verilog and then uses APR of standard cells and auxcells to generate the AMS layout. Leveraging the digital tool flow is more flexible and process portable than purely analog layout automation tools such as [3], which require more porting effort.

Auto-generating auxcell layouts is a key remaining gap in AMS synthesis that has until now remained a manual process. The number of required new auxcell functions varies between generators, depending on the specific analog component to be built using the generator. For example, a PLL uses two (e.g., tri-state buffer, switched capacitor), and a memory macro utilizes six auxcell functions [2] (e.g., sense-amplifier, word-line driver). Fig. 1(b) shows the impact and the usecase of

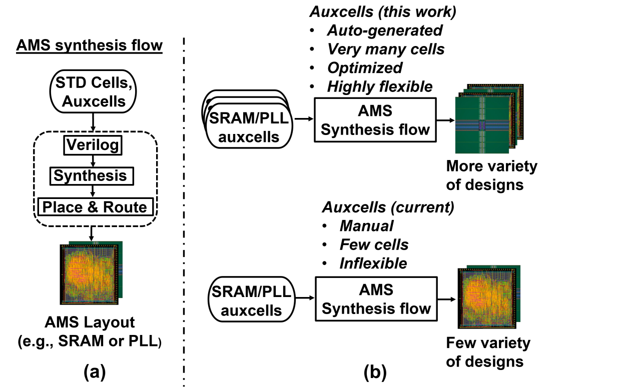


Figure 1: (a) The flow for synthesizing AMS circuits requires auxcells. This work replaces manual auxcell design and layout with auto-generated auxcells; (b) Usecase and the impact of the proposed framework on the AMS synthesis and component generators.

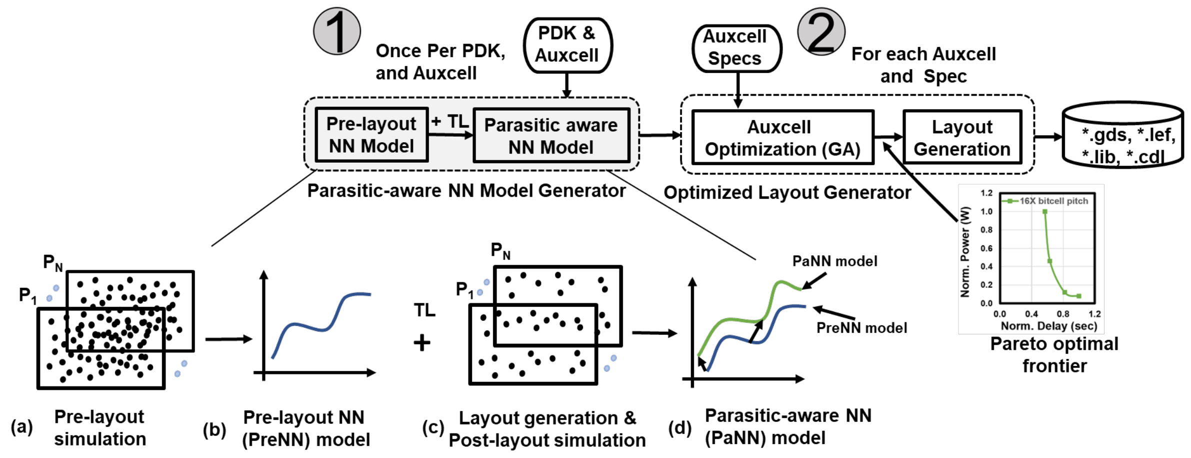
the proposed framework. Auxcells synthesized along with standard cells must match their height with the correct number of standard cell routing tracks (e.g., 7.5 or 9 tracks) in the standard cell library being used. For synthesizing memories, the auxcells must pitch-match with the bitcells by aligning with an integer number bitcell heights (for row-wise auxcells) or widths (for column-wise auxcells).

Auxcells should be parameterizable to span the vast design space of SoC design, enabling varying drive strength, device type, and different operating voltages. Manual auxcell design for a large set of cells is tedious and impractical, limiting both porting time and optimality of generated designs, which undermines the best trait of synthesizable analog generators.

To solve this problem, we propose a framework called auxiliary cell generator (AuxcellGen) that automates the generation of optimized and pitch-matched auxcell layouts. Functionally, AuxcellGen auto-generates auxcells by integrating: (1) a parasitic-aware surrogate model of auxcell behavior, (2) a circuit optimizer that uses the surrogate model to optimize each auxcell for its given user specification, and (3) a cell generator to auto-generate the pitch-matched auxcell layouts. To the best of our knowledge, AuxcellGen is the first framework that auto-generates pitch-matched memory auxcells.

II. Proposed Framework

In this work, we propose AuxcellGen, a technology-agnostic framework to auto-generate auxiliary cells for AMS synthesis flows [1] [2]. Fig. 2 presents a high-level overview of the framework and the critical steps in auto-generating the



1
Once Per PDK, and Auxcell
PDK & Auxcell
Pre-layout NN Model
+ TL
Parasitic aware NN Model
Parasitic-aware NN Model Generator
2
Auxcell Specs
For each Auxcell and Spec
Auxcell Optimization (GA)
Layout Generation
Optimized Layout Generator
*.gds, *.lef, *.lib, *.cdl
16X bitcell pitch
Norm. Power (W)
Norm. Delay (sec)
Pareto optimal frontier
P_N
P_1
TL
+
PaNN model
PreNN model
(a) Pre-layout simulation
(b) Pre-layout NN (PreNN) model
(c) Layout generation & Post-layout simulation
(d) Parasitic-aware NN (PaNN) model

Figure 2: High-level flow of the Proposed AuxcellGen framework.

auxcells. For even a simple auxcell function, (e.g., a tri-state inverter for a digitally controlled oscillator (DCO)), three transistors with different widths, lengths, fin-counts, etc. create a design space that is unwieldy to simulate exhaustively. For more complicated auxcells (e.g., sense-amplifiers), the design space is intractable for comprehensive simulation. To solve this, we use a surrogate model to emulate an auxcell's performance across the design space as a function of input parameters. Prior art [4] [5] demonstrated that NN-based modeling evaluates circuit performance fast and accurately versus SPICE simulation but does not consider layout parasitics, which we find are significant. We describe a parasitic-aware NN (PaNN) surrogate model that spans the design space and includes layout parasitic effects without losing accuracy.

As illustrated in Fig. 2, we generate the PaNN model for each auxcell type once-per-PDK by (a) simulating a sparse set of randomly selected points in that auxcell's design space using template based SPICE simulation, (b) fitting a pre-layout NN model (PreNN) to those pre-layout simulations, (c) simulating post-layout auxcells (generated by our cell generator) for a small subset of the pre-layout designs, and (d) using transfer learning to generate a post-layout, PaNN model for the auxcell type.

Once the PaNN model is generated for a given auxcell type, the framework can reuse the model to create optimized auxcell instances for any desired design point targets (right side of Fig. 2). The framework uses the model to select the optimized design of that auxcell type for a given design target. A genetic algorithm (GA) explores the auxcell design space with the PaNN model to select the optimal circuit parameters for a given specification. Finally, the cell generator auto-generates the auxcell layout for that particular instance of the auxcell, producing a layout (.GDSII), SPICE netlists (.cdl), logical (.lib, .db), and physical libraries (.lef), and output specification file (.json). Next, we discuss the framework steps in detail.

A. PaNN Model Generation

Creating a PaNN model from only post-layout simulations would be impractical, since they take twenty times longer than pre-layout simulations. So, to achieve a highly accurate PaNN model without a significant increase in training data generation time, we generate the PaNN model in a two-step approach by leveraging transfer-learning (TL). TL reuses a pre-trained model (source model) to accelerate the training of a new model (target model). A TL approach was used in [6] to create a parasitic-aware surrogate model by adding extra linear input and output layers on top of the schematic model, but only for a limited set of layout sizes. We propose a TL technique that considers the auxcell width and height as design parameters. As described in Fig. 2 (a–d), we first create a PreNN model and use it as a feature extractor for the PaNN model. We use a multilayer perceptron (MLP) as the regression model for both the PreNN and PaNN models, defined as

$$Y = f(\boldsymbol{W}\boldsymbol{X} + b) = f\left(\sum_{i=1}^{K} w_i x_i + b\right) \tag{1}$$

where $\boldsymbol{W} = [w_1, w_2, ..., w_N]^T$ and b are the K-dimensional weight vector, and the bias, respectively, learned from training. Defined in (2), $\boldsymbol{X} = [x_1, x_2, ..., x_K]^T$ is a K-dimensional vector of auxcell circuit parameters such as transistor width/length, device type, and layout size, and is unique to each auxcell.

PreNN model generation: For a given auxcell, the PreNN, $\boldsymbol{f_{pre}} : \boldsymbol{X_m} \rightarrow \boldsymbol{M_{pre}}$ maps the relation between the circuit parameters and the pre-layout circuit performance metrics ($\boldsymbol{M_{pre}}$). Here, $\boldsymbol{X_m}$ is the vector $\boldsymbol{X}$ without the layout parameters. To create the $\boldsymbol{f_{pre}}$ model, AuxcellGen generates the pre-layout training data by running SPICE simulations for randomly selected samples from the auxcell parameter space. Then the model is trained using the Keras Python library to minimize the mean squared error (MSE) loss function such that the model is accurate on the training data set and generalized well on the validation data set.

PaNN model generation: After developing the PreNN model $\boldsymbol{f_{pre}}$, the PaNN, $\boldsymbol{f_{pos}} : \boldsymbol{X} \rightarrow \boldsymbol{M_{pos}}$ maps the relationship between the auxcell parameters and the post-layout performance metrics ($\boldsymbol{M_{pos}}$). First, the post-layout training data is generated by: 1) selecting very few samples within the design space, 2) generating the layouts for each sample using the cell generator (detailed in section III), 3) extracting the parasitic netlist, and running post-layout SPICE simulations to measure the post-layout performance. Subsequently, $\boldsymbol{f_{pre}}$ is instantiated as the base model, and the post-layout training data is run through it to record the outputs as the metrics vector, $\boldsymbol{M_{pre}}$. The $\boldsymbol{M_{pre}}$ vector and layout input parameters (width and height of auxcells) are now the input features for the PaNN model. The PaNN model is created using these inputs in the same process as the PreNN model. Despite training with very few post-layout samples, the PreNN model achieves high performance accuracy, as the model is a mapping between the $\boldsymbol{M_{pre}} \rightarrow \boldsymbol{M_{pos}}$ instead of a mapping between high-dimensional input vector $\boldsymbol{X} \rightarrow \boldsymbol{M_{pos}}$.

The NN models are unique to each auxcell type, and no single model configuration works equally well for all auxcells. Furthermore, numerous hyperparameter choices, such as the number of hidden layers, nodes per hidden layer, and activation functions, can be set during the training. Determining the best model parameters and configuration for the given auxcell is an optimization problem. We implement Bayesian optimization (BO) [7] hyperparameter tuning to automatically determine the best parameters and obtain the high-performance NN models for the given auxcell. The entire process is automated using Python and a set of auxcell design templates.

B. Optimized Auxcell Netlist Generation

The task of design-space searching for the parameter set that satisfies the auxcell specifications can be generalized as a constrained nonlinear programming problem as below:

$$\begin{aligned} \underset{\boldsymbol{X}}{\text{minimize}} \quad & f(\boldsymbol{X}) = f(m_1(\boldsymbol{X}), m_2(\boldsymbol{X}), ..., m_N(\boldsymbol{X})) \\ \text{subject to} \quad & g_i(\boldsymbol{X}) = 0, \qquad\qquad i = [1, 2, ..., I] \\ & h_j(\boldsymbol{X}) \leq 0, \qquad\qquad j = [1, 2, ..., J] \\ & x_k^{(L)} \leq x_k \leq x_k^{(U)}, \;\; k = [1, 2, ..., K] \;\; \forall\, x \in \boldsymbol{X} \end{aligned} \tag{2}$$

where $f(\boldsymbol{X})$ is the objective function to be minimized, which is a function of N circuit performance metrics $m_n(\boldsymbol{X})$. The functions $g_i(\boldsymbol{X})$ and $h_i(\boldsymbol{X})$ formulate the equality and inequality constraints, respectively, that must be satisfied by the solution to the optimization formulation. In the case of the auxcell optimization problem, $m_i(\boldsymbol{X})$, $g_i(\boldsymbol{X})$, and $h_i(\boldsymbol{X})$ are the circuit performance metrics that must be minimized or satisfied and will be estimated using the auxcell PaNN model. For example, for sense-amplifier auxcell $m(\boldsymbol{X})$ is the power, $g(\boldsymbol{X})$ is the cell height and $h(\boldsymbol{X})$ is the delay to be satisfied. $x_i^{(L)}$ and $x_i^{(U)}$ define the lower bound and upper bound of the parameter $x_i \in X$, respectively, which collectively set the design space of the auxcell.

As defined by (2), auxcell optimization aims to minimize the cost function $f(\boldsymbol{X})$ by varying K parameters within the parameter range and evaluating the N metrics and $I \times J$ constraints using the PaNN model. For the auxcell design space search, we chose a GA for single-objective functions and NSGA-II for multi-objective functions, as their solutions are independent of the objective function type and robust to multiple constraints.

GA and NSGA-II are stochastic-based search metaheuristics which borrow inspiration from natural biological evolution, where fitter individuals survive. For given auxcell specifications, the GA starts by initializing the individuals of the population P of a predefined size $|P|$. In the generation t_i, the population P undergoes the fitness evaluation and offspring O generation through parental selection, crossover, and mutation, where the individual's fitness is calculated by evaluating $f(\mathbf{X})$ in (2). This process is repeated until the termination criterion T is met. NSGA-II follows the general outline of a genetic algorithm with a modified mating and survival selection. We advise the readers to refer to [8] [9] for more details on GA. **Constraints Handling:** We use a penalty parameter-less (PPL) approach [10] that does not require any penalty parameters or weights for handling the constraints. The cost function $F(\mathbf{X})$ in this approach is given by the formulation below, where f_{max} is the objective function value of the worst feasible solution in the population. The PPL approach not only results in better optimal solutions but is particularly viable to the frameworks like AuxcellGen, which has to auto-generate the different types of auxcells, and across a broad range of specifications, without the user figuring out the best penalty parameters or weights for handing the constraints.

$$F(\boldsymbol{X}) = \begin{cases} f(\boldsymbol{X}), & \textit{if feasible} \\ f_{max}(\boldsymbol{X}) + \sum_{i=1}^{I} g_i(\boldsymbol{X}) + \sum_{j=1}^{J} h_j(\boldsymbol{X}), & \textit{otherwise} \end{cases}$$

Once the optimized cell parameters are set, the design is passed to the layout engine for layout auto-generation.

III. Auxcell Layout Engine

We now describe automatic layout generation for auxcells. Based on a PDK abstraction for a given process, we define the concept of a unit cell and then develop parameterized layouts that implement transistor widths from the schematic with an appropriate number of unit cells. We use user-specified wider wires (or parallel wire connections in FinFET nodes) to reduce interconnect/via resistances. The auxcell height/width is selected considering geometric constraints, e.g., standard cell height matching for a tri-state buffer auxcell for PLL and bitcell pitch matching for memory auxcells.

A. Process Abstraction

A simplified process abstraction is vital to developing a process-portable and DRC-clean layout engine. Our cell generator uses abstracted grids, as described in [11], for both the FEOL and BEOL layers of a process to generate design-rule-correct layouts. These abstract grids are described in simple JSON files, allowing easy porting by simply modifying the grid entries in the JSON file and adding/removing new layers according to the process. More details can be found in [11].

B. Generating Auxcell Layouts

We define a library of commonly-used analog building-block cells (e.g., inverters, differential pairs (DPs), current mirrors, capacitor arrays, resistor serpentines, switches), or *primitives*,

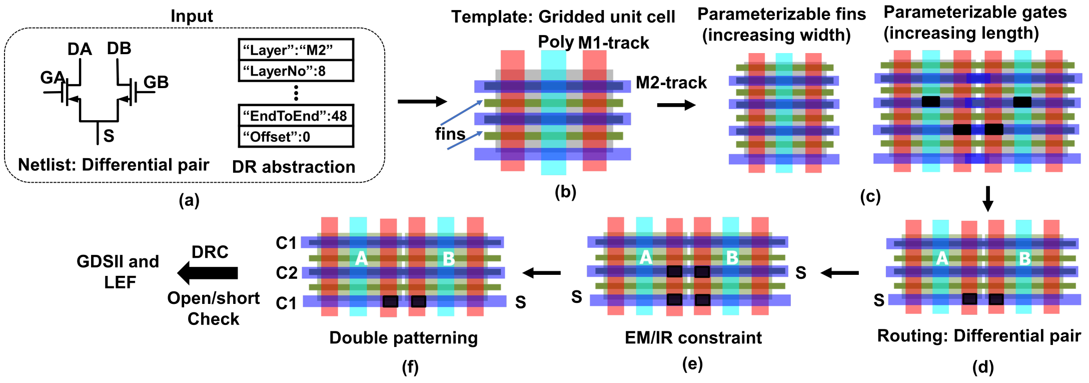


Figure 3: Illustration of the cell generator on a simple DP example.

where the connectivity of the primitive is defined using SPICE syntax, and a parameterized layout template for each primitive. Auxcells may consist of one or more primitives, e.g., a tri-state buffer for PLL and a sense-amplifier for a memory array that uses inverter and switch primitives. We will use the running example of a DP primitive cell, as shown in Fig. 3, to illustrate each step. The connectivity of the primitive is shown in Fig. 3(a): it consists of two transistors, A and B, with drain terminals DA and DB, respectively, gate terminals GA and GB, respectively, and a common source terminal S. The parameters of the abstracted grids for the PDK, which are used in the cell generator, are also shown.

1) Parameterized Unit Cells: A unit cell corresponds to a transistor of a certain effective width and length (with a parameterizable number of fins that define the effective width) and series-connected FinFETs (which define the effective length). The unit cell dimensions can be user-specified with a defined default value. In analog circuits, device sizes are often large and are built by combining multiple unit cells.

The unit cell layout is defined by a layout template, illustrated in Fig. 3(b). This template defines the features that make up the cell, e.g., poly gate (cyan), fins (green), M1 (red), and M2 (blue) metal tracks, defined on the process-independent DR abstraction grids. The layout template interacts with the DR abstraction (JSON file) to determine feature widths and grid spacings to generate the layout of the unit cell, and the use of gridding guarantees correct-by-construction DRC-correct layouts. The M1 grids also define transistor terminal locations for the cell routing module to connect the terminals.

The layout of a schematic transistor can then be parameterized according to the designer's needs, using the unit cell as a building block. Fig. 3(b) shows a unit cell template with three fins, two M1 tracks, and three M2 tracks. This is used to build the two parameterized cells shown in Fig. 3(c), both parameterized to 5 fins and 4 M2 tracks, with the one at right also parameterized to use a gate length of $2L_{min}$.

2) Primitive cell layouts: The cell generator automatically creates layouts for primitive cells (inverters, DPs, current mirrors, differential loads, capacitor arrays, resistors, and switches) which are constructed from unit cells; the auxcells, in turn, are built using primitives.

Aspect ratio: The cell generator is parameterized to generate primitives in different aspect ratios. However, the aspect ratio of primitives is determined from geometry constraints from designs, e.g., in a tri-sate buffer auxcell for PLL, the heights are fixed to match standard cell row height, and in memory designs, the width is selected based on bitcell pitch.

Placement: Based on a user-specified placement scheme that includes geometric constraints (common-centroid, symmetry, matching), the unit cells that make up a transistor are placed in an array on the DR abstraction grid to achieve the specified aspect ratio(s). For example, in a sense-amplifier auxcell for memory design, inverter and footer switch primitives should be placed symmetrically, and NMOS/PMOS transistors of the two inverter primitives must be matched to minimize offset. Furthermore, NMOS (PMOS) transistors of inverters, in a sense-amplifier auxcell can be placed in common-centroid fashion for matching, using a method based on [12]. For example, when the size of each transistor, A and B, in Fig. 3 corresponds to one unit cell, a total of two unit cells are required for the DP, and these cells are placed as shown in Fig. 3(d).

Cell routing: Based on the netlist connections, routing is performed as follows: as the M1 and poly grids (i.e., S/D/G terminals) of all devices are exposed, first, an M2 track is selected to connect the terminals, and thereafter, vias are dropped at the intersections to M1 (i.e., at S/D grids). For the DP in Fig. 3, vias are dropped at the first M2 track to the sources of A and B, as shown by the black squares in Fig. 3(d). M2 tracks are used to connect transistors in a row, and M3 tracks, orthogonal to M2, are used to connect transistors across rows. Low-resistance connections, implemented as parallel wires in FinFET technologies due to width quantization, can be used as shown in Fig. 3(e) which uses two wires to connect the source nodes (black vias in the two lower M2 tracks).

Coloring rules: Other than enforcing basic DR checks, handled by PDK abstraction, the cell generator also respects coloring rules for metal layers. This is shown in Fig. 3(f) where the colors C1 and C2 for the M2 layer are alternately assigned to tracks. Finally, DRC-clean GDSII and LEF files are generated.

IV. Results and Discussion

We now present the results of AuxcellGen by auto-generating a PLL tri-state buffer and an SRAM sense-amplifier.

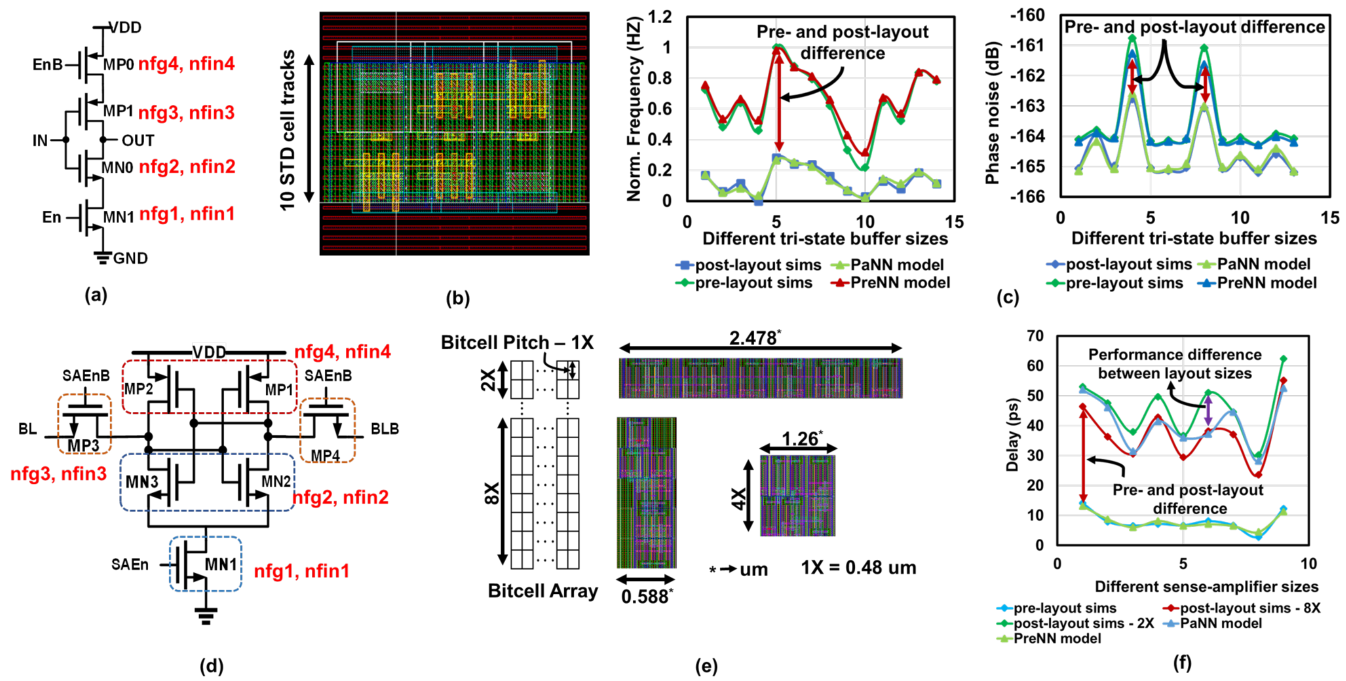


Figure 4: (a) Schematic of the tri-state buffer auxcell. (b) Standard-cell-compatible layout of the tri-state buffer. (c) The frequency and phase noise results comparison between the pre-layout simulations, post-layout simulations and NN models. (d) Schematic of the sense-amplifier auxcell. (e) Layouts of the sense-amplifier auxcell with 2X, 4X, and 8X bitcell pitches. (f) Sense-amplifier delay comparison between the pre-layout simulations, post-layout simulations, and NN models.

A. *PaNN modeling and Layout Generation:*

The PaNN model training data for the two example cells was generated by running SPICE simulations across three 16-core CPU machines with 16-parallel jobs per machine. The data is split into 60:40 ratio between training and testing data sets.

Tri-state buffer: Figure 4(a) shows the schematic of the tri-state buffer. Following the steps outlined in section II-A, the PaNN model is created which maps eight design parameters (the number of fins (nfin) and the number of fingers (nfg) for each transistor) to four design metrics (frequency (Freq), phase noise (PN), figure of merit (FoM), and power). The PreNN model is created with 2500 design samples and the PaNN model with 100 post-layout samples. The hyperparameter tuning with BO mentioned above resulted in the 3-hidden layer MLP with 200 nodes per layer, ReLU activation function, and 0.394% testing MSE loss. With parallel runs, it took 2.6 hours to generate pre-layout training data, with each sample taking 5 minutes, and 1.04 hrs for post-layout training data, with each sample taking 30 minutes. The layout engine generates the layouts, with each sample taking only 15 seconds for the complete layout. Figure 4(b) shows the tri-state buffer layout for the 10-track STD cell height, and Figure 4(c) demonstrates the frequency and phase-noise results for various tri-state buffer designs. It is evident from the figure that there is an average difference of 15–30% between the pre-layout and post-layout simulation results, confirming the importance of including layout effects for accuracy. Moreover, the figures show that the proposed TL-based parasitic-aware NN model closely matches the results from simulations.

Sense-amplifier: Figure 4(d) shows the schematic of the SRAM sense-amplifier auxcell with seven transistors and eight nfin/nfg design parameters. The PaNN model is created by mapping these eight parameters to three design metrics (offset, delay, and power). The PreNN model is created with 2000 design samples and the PaNN model with 240 (60 samples for each 2X, 4X, 8X, and 16X bitcell pitch) samples. The hyper-parameter tuning with BO generated a 2-hidden layer MLP with 198 nodes per layer, ReLU activation function, and 1.84% testing MSE loss. The parallel jobs took 10.25 hours to generate pre-layout training data, with each sample taking 15 minutes for 1000 Monte-Carlo (MC) points and 22.5 hrs for post-layout training data, with each sample taking 4.6 hours. The layout engine generates the layouts, with each sample taking only 45 seconds for the complete layout. Compared to the manual layout, the 4X auto-generated layout differs by 6.9% and 4.2% in the area and the performance, respectively. The area difference is primarily due to the placement of the primitives at the top level. With oxide sharing between primitives, this difference will be further reduced significantly, yielding auto-generated layouts close to the manual layouts in terms of area and performance. Figure 4(e) shows the sense-amplifier layouts for 2X, 4X, and 8X bitcell pitches. The Figure 4(f) demonstrates the delay results for various sense-amplifier designs with different bitcell pitches with an average difference of 35–50% between the pre-layout and post-layout simulation results. More importantly, the delay value for the same design but with a different bitcell pitch varies drastically. Therefore, it is essential to create the PaNN model with layout size (bitcell pitch for the sense-amplifier) parameters and capture the parasitics across the layout sizes. The figure depicts that a single sense-amplifier PaNN model predicts the delay across the bitcell pitches with accuracy close to the post-layout simulations, re-iterating the efficiency of the proposed TL technique-based PaNN model generation.

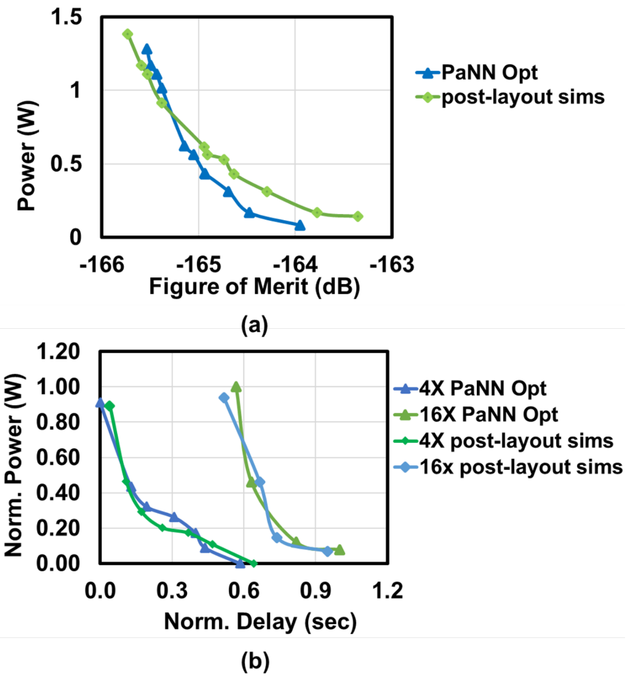


Figure 5: (a) The Pareto optimal frontiers of the PLL Tri-state buffer (b) The Pareto optimal frontiers of the SRAM sense-amplifier for same design constraints, but with different bitcell pitches

B. Auxiliary cells optimization

Once the auxcell parasitic-aware NN models are created for the given PDK and the auxcell type, the framework is ready to generate the auxcells for various user-specified specifications. We use the pymoo framework [13] to implement the GA and NSGA-II algorithms. Based on our experiments, an initial population size of 100 with stopping criteria of 400 generations can generate optimal results for the auxcells.

Tri-state buffer Optimization: The optimization goal for the PLL tri-state buffer auxcell is to minimize the power (P) and maximize the FoM with constraints on the frequency ($Freq$), phase-noise (PN), and cell height (H). The objective function is $f(\boldsymbol{X}) = w_1 P(\boldsymbol{X}) - w_2 FoM(\boldsymbol{X})$ (The negative sign in the second term is used because FoM is negative and is to be maximized), subject to $Freq(\boldsymbol{X}) \geq Freq_{target}$, $PN(\boldsymbol{X}) \leq PN_{target}$ and $H(\boldsymbol{X}) = H_{target}$ constraints, where w_1 and w_2 are the weight coefficients. For $Freq_{target} \geq 5$ GHz, $PN_{target} \leq 128$ dB, and $H_{target} = 10$ units, the Pareto-Optimal frontier (POF) of the AuxcellGen-generated tri-state buffer is shown in Fig. 5(a).

Sense-amplifier Optimization: The optimization goal for the SRAM sense-amplifier auxcell is to minimize the offset (Off) and power (P) with constraints on the delay (D) and cell height (H). The objective function for the auxcell is $f(\boldsymbol{X}) = w_1 P(\boldsymbol{X}) + w_2 Off(\boldsymbol{X})$, subject to $D(\boldsymbol{X}) \leq D_{target}$ and $H(\boldsymbol{X}) = H_{target}$ constraints. For $D_{target} \leq 10$ ns and $H_{target} = 4$ units, the Pareto-Optimal frontier (POF) of the AuxcellGen-generated sense-amplifier is shown in Fig.5(b). The experiment is repeated by changing the $H_{target} = 16$ units, and new POF can be observed as shown in Fig. 5(b).

To measure the accuracy of the proposed optimization using the PaNN model, we generated the layouts for each POF point using the cell generator, and the performance was measured by running the post-layout simulations. As shown in Fig.5 the post-layout simulation results for both the auxcells match closely with optimization results, with an average error of 6.4% for the tri-state buffer auxcell and an average error of 7.2% for the sense-amplifier.

V. Conclusion

In this paper, we proposed a framework to auto-generate the optimized layouts for the analog and memory auxcells. We developed a TL-based parasitic-aware modeling technique to model the layout parasitics considering the layout sizes effectively. The framework fills the gap in the AMS synthesis flow and enables the generation of a wide variety of AMS designs.

VI. Acknowledgement

This work was funded in part by Defense Advanced Research Projects Agency (DARPA) under agreement no. FA8650-18-2-7844 and as part of the ALIGN project, under NIWC Contract N660011824048.

References

[1] T. Ajayi, *et al.*, “An open-source framework for autonomous SoC design with analog block generation,” in *Proc. IFIP/IEEE VLSI-SOC*, pp. 141–146, 2020.

[2] S. Kamineni, *et al.*, “MemGen: An open-source framework for autonomous generation of memory macros,” in *Proc. CICC*, pp. 1–2, 2021.

[3] B. Xu, *et al.*, “MAGICAL: Toward fully automated analog ic layout leveraging human and machine intelligence: Invited paper,” in *Proc. ICCAD*, pp. 1–8, 2019.

[4] G. İslamoğlu, *et al.*, “Artificial neural network assisted analog ic sizing tool,” in *Proc. SMACD*, pp. 9–12, 2019.

[5] G. Wolfe and R. Vemuri, “Extraction and use of neural network models in automated synthesis of operational amplifiers,” *IEEE T. Comput. Aid. D.*, vol. 22, no. 2, pp. 198–212, 2003.

[6] J. Liu, *et al.*, “From specification to silicon: Towards analog/mixed-signal design automation using surrogate nn models with transfer learning,” in *Proc. ICCAD*, pp. 1–9, 2021.

[7] P. I. Frazier, “A tutorial on Bayesian optimization,” *arXiv preprint arXiv:1807.02811*, 2018.

[8] G. Alpaydin, *et al.*, “An evolutionary approach to automatic synthesis of high-performance analog integrated circuits,” *IEEE T. Evol. Comput.*, vol. 7, pp. 240–252, 2003.

[9] R. Martins, *et al.*, “LAYGEN II—automatic layout generation of analog integrated circuits,” *IEEE T. Comput. Aid. D.*, vol. 3, no. 1, pp. 1641–1654, 2013.

[10] K. Deb, “An efficient constraint handling method for genetic algorithms,” *Computer Methods in Applied Mechanics and Engineering*, vol. 186, no. 2, pp. 311–338, 2000.

[11] T. Dhar, *et al.*, “ALIGN: A system for automating analog layout,” *IEEE Des. Test*, vol. 38, no. 2, pp. 8–18, 2020.

[12] A. K. Sharma, *et al.*, “Performance-aware common-centroid placement and routing of transistor arrays in analog circuits,” in *Proc. ICCAD*, pp. 1–9, 2021.

[13] J. Blank and K. Deb, “pymoo: Multi-objective optimization in python,” *IEEE Access*, vol. 8, pp. 89497–89509, 2020.