SeMicro-PR 2024

A Simulation-based Python EDA Tool for Customized Digital Cell Design

R. Journois[1,2], L. Compassi-Severo[1], O. Saotome[1]
[1]Aeronautics Institute of Technology, São José dos Campos, Brazil
[2]CentraleSupélec, Gif-sur-Yvette, France
{robinson, severolcs, osaotome}@ita.br

***Abstract* - With the increase of computation capability, new optimisation methods are becoming accessible. As transistor size reduction is less and less effective and more complex with each process node, digital design needs to get more efficient. One way to do that would be to use analog design methods. Previous works have presented ways to automatise analog circuit design and allow automatic exploration of high dimensional parameter space. Digital cells have to follow certain restrictions on shape and size in order to be properly integrated in large-scale circuit. The use of automation with digital cell design will allow the creation of custom libraries and circuits faster, reduce area and as such, cost. This work presents a Python script which links minimisation methods, circuit simulation, behaviour analysis and digital cells constraints to allow netlist level parameter optimisation.**

I. INTRODUCTION

Recent years have seen the development of electronic design automation (EDA) tools focused on analog integrated circuit (IC) design with the help of deterministic and heuristic optimisation algorithms. Rao et al. have used a mix of genetic algorithm and simulated annealing on the design of a voltage-controlled oscillator [1], White et al. showcase an inverter and buffer design optimisation with particle swarm [2] and Martins et al. have built a full-fledged design automation software, AIDA [3].

As Moore's law is becoming a marketing ploy rather than the transistor size reduction indicator it once was [4], improvements in digital circuits efficiency will increasingly stem from better design paradigms, both in terms of inter-cell routing and cell design. Digital cells usually come from libraries which are built for generic use cases. However, custom ICs can have specific requirements on power, voltage, frequency or radiation shielding, which requires specific cells that can be quite different from the usual libraries. Building a cell library requires specialised knowledge and as such, is time-consuming and costly. The use of tools that adapt analog design methods to fine-tune cells to their specific use case might make digital circuits better with only a few additional design steps. Moreover, with the improvement of computing capabilities, algorithm flow analysis, and notably with stochastic processes, the use of routing algorithms allows non-standardised cells and large-scale system optimisation. As such, the use of automatic cell generation might enable digital designs that are more area-efficient and better adapted to their use case.

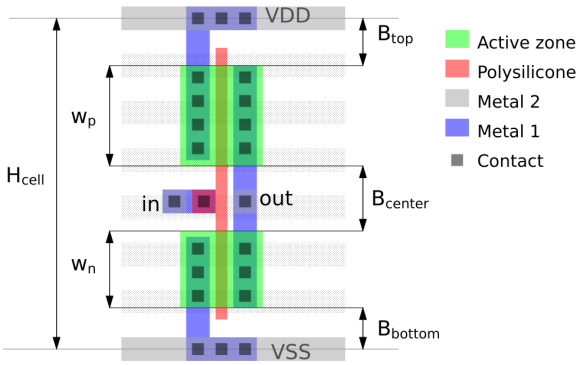

Fig. 1: *Schematic of a CMOS inverter layout with a height of 7 metal tracks.*

This work introduces a Python-based framework to optimise the parameters of integrated circuit cells. The proposed framework is demonstrated with the design of inverter cells using a CMOS 65 nm process with different cell heights in number of metal tracks and target specification for optimisation.

II. METHODOLOGY

In order to build an efficient optimiser, one has to ponder on several issues. Before even thinking about the structure of the program, it is necessary to define the optimisation's objectives and constraints on the parameters, how to incorporate this in the cost function, how to evaluate it, and choose the right optimisation method. After explaining this, this section tackles the program in itself, the programming philosophy and the structure of the script along with an example.

A. Constraints and objectives

When optimising a system, one has to define the objectives and constraints. Constraints can be imperative, meaning that no solution will exist without satisfying them, or suggestive, meaning that solutions will try to get as close as possible to respecting them but might generate solutions that do not. In our case, imperative constraints will be voltage and spatial dimensions. Suggestive constraints can also be optimisation objectives and will be incorporated in the cost function.

Area minimising is always important for optimised design, but all dimensions are not free. Digital standard cells usu-

ally have a height limitation (H_{cell}) defined by the number of metal tracks used for routing. Figure 1 shows an example of CMOS inverter cell with a height of 7 metal tracks. The value of H_{cell} should include the necessary space for the interconnections, bulk biasing if used, NMOS and PMOS channel width (w_n and w_p), as well as the distances required by the fabrication Process Design Kit (PDK) rules. In Fig. 1, the vertical spaces would be B_{top}, B_{center} and B_{bottom}. This means that, depending on layout options and with w_n, w_p the widths of an NMOS and a PMOS, we have an upper limit on $w_n + w_p$ for each CMOS pair. Based on that, the main constraint of the Inverter cell design is given by:

$$w_n + w_p < H_{cell} - \min(B_{top} + B_{center} + B_{bottom}) = H_{max} \quad (1)$$

This equation holds for all cells with the same topology. Other constraints might include maximum delay, static trip points, leaking power or even cell length - the dimension perpendicular to transistors - as a crude pre-layout way to evaluate the area.

B. Test bench

Test-benches used by the algorithm are such as showcased in figure 2. As the goal is to evaluate the cell in working conditions in order to evaluate the circuit specifications, its outputs should be connected to some real load condition. In our case, a CMOS inverter to represent a fanout of 1.

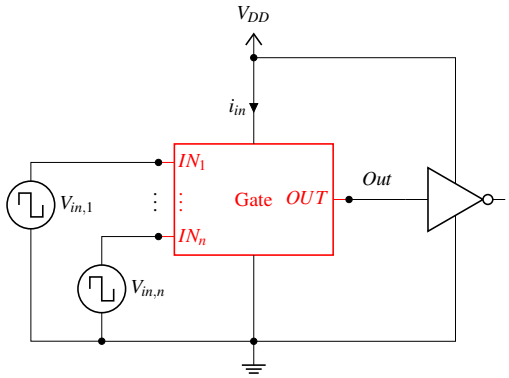

Fig. 2: *Schematic of the test bench for the evaluation of a gate's performance.*

C. Cost function evaluation

As one will have several constraints to respect and variables to evaluate, the construction of the optimisation metric is quite important. Trip-point equilibrium, delay and power measurement have been implemented in Python. These evaluations are then congregated to obtain the scalar cost function that the optimisation algorithm will try to minimize. Since this function is an aggregate of aggregated simulated results, it is not expected to be differentiable or otherwise present any regularity.

D. Black box optimisation

Because of the possible irregularity of the cost function, the use of differential and convex optimisation methods is not ideal, which is why this program uses stochastic black-box optimisation algorithms. This means that we are not able to ensure that we have reached the best possible set of parameters. Hence, exploring the whole design space and avoid convergence on local minima is important. The first implemented method is a simple genetic algorithm as showcased in [1] but future work will focus on implementing and comparing several other optimisation methods.

E. Building for modularity

In order to compare several optimisation methods and metrics and to increase reusability of the code, this program needs to be modular. It should be possible to individually change the simulator, type of simulation and measurement, type of optimisation and type of cell without having to rewrite other things. For example, this program is built around Cadence's Spectre for simulation, but one might want to use a physics simulator for some components, or a previously generated circuit behaviour look-up table, which is feasible by providing a Python interface. It should also be possible to use only a part of the program to run parameter space exploration, run the same simulation with different programs, or even run the simulations and the optimisation on different computers.

F. Program Outline

The first version of the program is functional programming based, as showcased in the following figure 3.

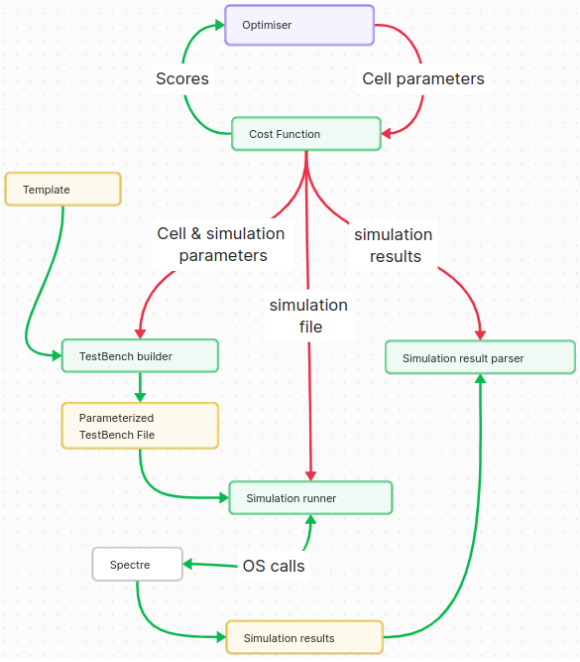

Fig. 3: *Functional program graph, arrows: function calls in red, inputs and outputs in green, cells: files in yellow, functions in green and main loop in violet.*

This generic version is the basis of our work, but we will go into the details with the example shown in figure 4. This graphic presents the minimisation of the delay for an inverter cell of height H_{cell} through the variation of the widths of its NMOS and PMOS transistors. The delay τ is defined as follows:

$$\tau = \frac{\tau_{rise} + \tau_{fall}}{2} \tag{2}$$

where for $i \in \{rise, fall\}$:

$$\tau_i = t|_{V_{In}=VDD/2} - t|_{V_{Out}=VDD/2} \tag{3}$$

The inverter is the same as in A, it is composed of NMOS and PMOS transistors with w_n and w_p their width with the relation $w_n + w_p \leq H_{max} < H_{cell}$, which means the only variable of this problem is either w_n or w_p. The script calls the optimiser by setting the name of the parameter or set of parameters to be optimised, w_n in our example. During the optimisation loop, the optimizer decides a set of parameters describing a cell that needs to be tested, let's say $w_n = w_{n,min}$ as specified by the PDK, and calls the cost function, here `delay_evaluation`. In turn, the cost function calls the test-bench builder with the cell and simulation parameters in order to create the `.scs` netlist, in our case the cell parameters would be $w_n = w_{n,min}$ and $w_p = H_{max} - w_n$ and simulation parameters would be its type, transient, and the final time of the simulation. Once the test-bench builder has finished its process, the cost function calls the simulation runner, which uses Python's `os`'s library and takes in the previously generated `.scs` file and outputs the results in the form of `.psf` ASCII file. The parser is then called to extract the signals for the cost function to process. In our case, it would be the output and inputs voltages to evaluate delays. The optimiser runs the process again until sufficient convergence has been reached. File interaction and directory creation are done through a configuration file that defines the paths. A data-holding capability is incorporated in the algorithm in order not to run the same simulation twice. It also allows for the plotting of graphs such as in figure 6.

III. RESULTS

The algorithm being implemented, a few things are worth noting. This part tackles which tools the program was run with, then discusses the importance of the simulator precision, the limits of the genetic algorithm along with an example of what can be done with such a script.

A. Tools

This work is built with Cadence's Spectre as a simulator but not around it. It could be used with other simulation software, such as a physics simulator or a cell behaviour look-up table, though the user would have to provide the Python interface. This simulator was chosen for its wide usage and particularly because it is the software of choice in our team which makes it easy to integrate in our design workflow. We built a Spectre 21.1 Python interface using `os` library. Provided spectre command line syntax does not change, it should work for other versions, but the user is free to plug in another API. The computer used for this work is a shared server with a

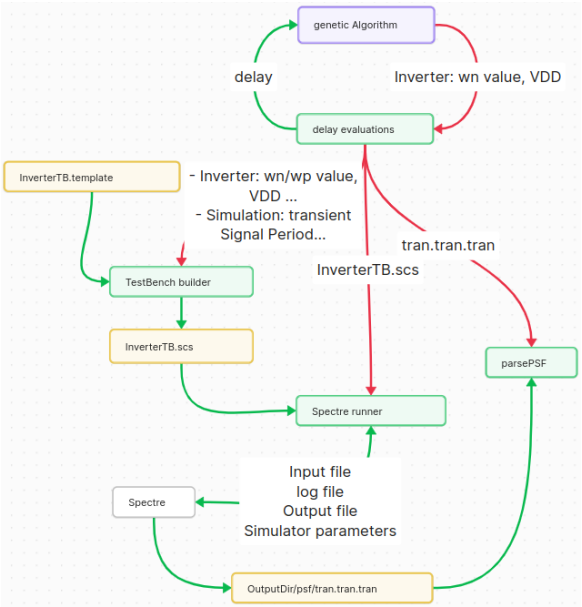

Fig. 4: *Functional program graph for delay optimisation of an inverter, arrows: function calls in red, inputs and outputs in green, cells: files in yellow, functions in green and main loop in purple.*

32-core 13th gen Intel CPU and 128 GB of RAM running Rocky Linux 8.9 for Cadence's suit compatibility. For simple dependency management, a Python virtual environment running Python 3.12 is used, as it is one of the latest versions supported by the server's distribution.

B. Simulator Precision

Since the signal processing is done in Python, the simulated signal resolution is quite important, especially when derivatives or y-value matching are involved. With the same example as before and with the definition of the delay τ in equation (2), getting a precise value of the time at which the signal is at $\frac{VDD}{2}$ is quite important. The first method used was linear interpolation between the two closest points. However, the resulting plot is shown in figure 5. Since the delay should be way more regular because of the simplicity of the system, we added a simulator side interpolation of the values at a higher number of points per signal period, which led to better results, and allowed us to perform optimisation as shown in 6.

C. Genetic Algorithm

We used `scipy`'s optimisation library for our optimisation algorithm with the `differential_evolution` function, which implements a genetic algorithm adapted for numeric parameters. Staying with the same example as in the previous sections, one can get a figure such as Fig. 6. The points that were explored by the algorithm are in black and the minimal value encountered is in red. As it is a stochastic algorithm, the result obtained is not optimal, but the algorithm stops when a sufficient number of values are found close to each other. On the aforementioned graph, one might expect better values in the non-explored part between 700 nm and 800 nm,

TABLE 1: Table showing the best performance and associated width for the delay, DC power and trip point equilibrium depending on cell height as number of tracks.

Cell Height	Delay		DC Power		Trip point at $V_{DD}/2$
($\times$ Metal Track)	w_n (nm)	delay (ps)	w_n (nm)	power (pW)	w_n (nm)
6	630	12.603	625	93.78	470
7	755	12.605	790	108.3	555
8	870	12.613	905	122.7	635
9	950	12.630	970	144.1	710
10	1140	12.653	1220	151.6	785
11	1210	12.630	1340	165.3	860

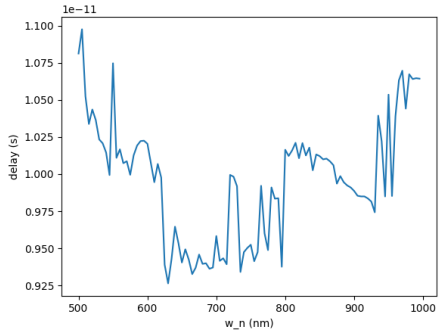

Fig. 5: *Delay sweep across w_n, $w_p = H_{max} - w_n$ with only Python level linear interpolation.*

but the algorithm stopping condition was reached. This is why the combination of global exploration methods such as genetic algorithms with local exploration such as simulated annealing is quite interesting as shown in [1].

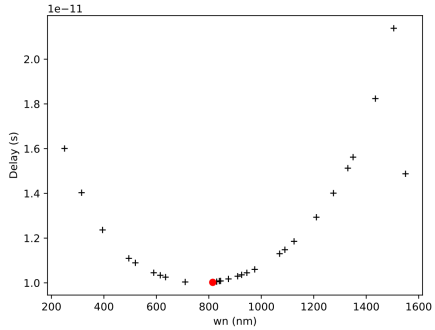

Fig. 6: *Cost function value for a delay minimisation*

D. Comparing Cell heights

As an example of the capabilities of the algorithm, Table 1 presents for different cost functions, the best parameter and simulated result for different cell heights with the same voltage. If cell height has a noticeable influence on DC power consumption, one might notice that delays do not change much, to the point that the variations might come from the aforementioned optimising algorithm uncertainties. If for smaller cells, optimal w_n for DC power and delay are somewhat close, it is always quite distinct from the value needed for DC trip-point equilibrium.

IV. CONCLUSION

This work presents a method to automate netlist level circuit dimensioning with criteria such as delay, DC power and behaviour and circuit dimensions as optimisation objectives. The program structure can be reused to build a simple cell library and explore circuit parameters space.

With it, we studied the variations of the best parameters of an inverter cell across several cell heights.

The use of the Python language, way more accessible than Cadence's SKILL language, makes it easier to use for new Cadence users. It allows for easier combination with other simulators, and facilitates the integration of various optimisation methods.

Future work includes integrating other cells, automatically generating the layout, and integrating post-layout simulation into the optimisation process.

Acknowledgments

This work was carried out with the partial support of the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) with process numbers 132114/2023-2 and 420693/2023-8.

REFERENCES

References

[1] V. V. Rao and I. Savidis, "Multi-Objective Simulation-based Optimization of Analog Transistor Sizing," Mar. 17, 2020.

[2] L. White, L. While, B. Deeks, and F. Boussaid, "Transistor Sizing Using Particle Swarm Optimisation," in *2015 IEEE Symposium Series on Computational Intelligence*, Dec. 2015.

[3] R. Martins, R. Lourenço, A. Canelas, R. Póvoa, and N. Horta, "AIDA: Robust layout-aware synthesis of analog ICs including sizing and layout generation," in *2015 International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*, Sep. 2015.

[4] "IRDS™ 2021: More Moore - IEEE IRDS™." (), [Online]. Available: `https://irds.ieee.org/editions/2021/more-moore` (visited on 11/29/2024).