Department of Physics and Astronomy
Heidelberg University

Bachelor Thesis in Physics
submitted by

Kaspar Frieder Haas

born in Karlsruhe (Germany)

2023

Parameterized circuits for accelerated design and simulation

This Bachelor Thesis has been carried out by Kaspar Frieder Haas at the
Kirchhoff Institute for Physics in Heidelberg
under the supervision of PD Dr. Johannes Schemmel

Abstract

In the development of increasingly complex analog neuromorphic chip designs, the use of flexible and scalable design parts can be of great advantage. This is because chips created using these so called Parameterized Cells (PCells) can become more flexible, while at the same time opening the possibility for wider use cases of these cells.

The work presented here demonstrates this by creating a scalable synapse driver array for use in the BrainScaleS-2 (BSS-2) system. Here the creation process of the array, including the creation of a second PCell, an address decoder, as well as some simulations will be shown. The results produced can even be seen as a first step towards the creation of a fully scalable BSS-2 core. To achieve this the described parameterization process has to be expanded to the other parts of the chip. Furthermore the simulations performed using the created synapse driver array show a possible speed gain when using the flexibility of PCells, by comparing the simulation times of the differently instantiated arrays to each other.

Kurzfassung

Bei der Entwicklung immer komplexerer analoger neuromorpher Chips kann die Verwendung von flexiblen und skalierbaren Designteilen von großem Vorteil sein. Die Gründe hierfür sind, dass die Chips, welche mit den parametrisierbaren Zellen erstellt werden, selbst flexibler werden und dass die parametrisierten Zellen in mehr Anwendungsfällen verwendet werden können.

Die vorliegende Arbeit zeigt dies anhand der Erstellung eines skalierbaren Synapsen Treiber-Arrays, welches im BrainScaleS-2 (BSS-2) System verwendet wird. Hierbei werden der Erstellungsprozess des Treiber-Arrays, der auch die Erstellung eines parametrisierbaren Adressdekoders enthält, sowie die durchgeführten Simulationen des Treiber-Arrays präsentiert. Weiter kann die Arbeit als erster Schritt in Richtung eines vollständig parametrisierbaren BSS-2-Kerns gesehen werden. Um diese vollständige Parametrisierung zu erreichen, müsste der hier vorgestellte Parametrisierungsprozess auf die anderen Chipteile angewendet werden. Darüber hinaus zeigen die durchgeführten Simulationen mögliche Zeitgewinne abhängig von der gewählten Parametrisierung des Arrays. Diese werden durch einen Vergleich von unterschiedlich parametrisierten Synapsentreiber-Arrays dargestellt.

Contents

1 Introduction

Modern microelectronics and electronic circuits have been experiencing an explosive growth over the last decades (Razavi, 2013, p. 1). This includes the development of many different kinds of electronic devices, for example cellphones, digital cameras or computers. One thing all these devices have in common is the fact that without the rise and rapid development of integrated circuits, or so called ICs, none of them would be able to exist in their current form. As developments progressed, these ICs became larger and larger, enabling the creation of faster and more powerful electronic devices. An example often used to demonstrate this massive growth is Moore's law, stating that the complexity of minimum cost components has and will continue to increase with a factor of two per year (Moore, 2006), resulting in a doubling of transistors used in integrated circuits every year. Comparing this statement with the actual number of transistors per microchip, growing from just over 2000 in 1971 to 58.2 billion in 2021 (Our World in Data, 2023), one can easily see that Moore's prediction was quite accurate. This also meant that the tools used to create integrated systems had to change over time. Nowadays most of VLSI, or Very-Large-Scale Integrated Circuit design is done using software like the design suite provided by Cadence®. Cadence® is one of the key leaders when it comes to electronic system design, cooperating with manufacturers like TSMC or UMC.

As these circuits grow in size, the design process is expected to become more and more complex too. As a result it is desired to break down the different parts of a design into smaller chunks which then can be used as building blocks or so called cells in larger designs. When creating these cells with future projects or possible changes in mind it only seems plausible to wish for Parameterized Cells, so that the created cell is flexible or scalable and can be reused in different designs. A possible way of creating these flexible cells is the creation of so called PCells in the design suite of Cadence®, Virtuoso®.

This work shows one possible implementation of such a scalable design in analog neuromorphic systems, by using the synapse driver array of the BrainScaleS-2 system as an example. To achieve the mentioned scalability a parameterized address decoder has been created as a PCell using the programming language SKILL. With this address decoder the old driver design was modified, now allowing the user to choose a configurable number of driver outputs.

The second part of this work gives an introduction into physical parasitic effects that must be taken into account when creating and simulating analog circuits. Here an extraction and simulation of the driver array in two different configurations will be performed, showing the difference in speed and accuracy when working with increasingly complex designs.

2 Background

The following sections will give an introduction into the topics covered within this thesis and will also mention some resources for further reading. After a short introduction into microelectronics and CMOS factoring some information about neuromorphic hardware with focus on the BrainScaleS-2 (BSS-2) system will follow. Then the concept of VLSI design using Virtuoso® and creating PCells for this application will be explained. The introduction will close on information about parasitics in chip design as well as hint at simulation uses of these parasitics.

2.1 Microelectronics and CMOS production

The rise of electronic devices started about a 100 years ago, arguably with the invention of the vacuum tube in 1904. This evolution, driven by the extended use of vacuum tubes and later semiconductors, for example in amplifying circuits, changed the world. While in the beginning this development seemed quite slow, it picked up speed in the 1940s with the discovery of the transistor by Shockley, as well as Brattain and Bardeen, working in the same research group (Riordan, Hoddeson, and Herring, 1999). With this discovery the foundation for modern semiconductors was laid. Even then, it was not until the 1960s that the microelectronics and especially ICs really took off (Razavi, 2013). One reason for this was that early ICs only contained a handful of devices. This changed rapidly with the invention of the CMOS process in 1967 (Wanlass, 1967), which allowed for larger and more complex designs to be created.

Talking about the use and implementation of transistors in these ICs the question "What exactly is a transistor?" may arise. Essentially there are two types of transistors: The so called bipolar transistor and the metal-oxid-semiconductor transistor (MOS) (Phillip E. Allen, 1987, p. 29). While the bipolar transistor has been the cheaper transistor for many years, the MOS transistor has fallen in costs so dramatically, that today it is widely used in all kinds of applications. Because of this widespread use and the fact that this work does not touch electronics with bipolar transistors, this introduction will only focus on the MOS transistors.

Simply put, a transistor can be seen as a controlled current source. Talking about MOS transistors, this control is achieved by controlling an electric field using a voltage. To generate this field different doped silicon layers, n+ and p+, are used. If these doped silicon layers, together with an oxide insulator, are placed in the right geometry, for a PMOS two p regions placed inside an n substrate as seen in figure 1, they form two pn-junctions between the n doped substrate and the p doped source and drain contacts.

Additionally, it can be seen that the gate and the substrate are forming the parallel plates of a capacitor, allowing for the construction of an electric field between these two contacts (Phillip E. Allen, 1987, p. 49). This electric field can change the sizes of the depletion regions created by the pn-junctions, allowing for the regulation of a current flowing from drain to source. In conclusion this means, that a voltage between the substrate, in figure 1 still called source, and the gate can be used to control the current flowing through the transistor. When using a PMOS as shown in the schematic in figure 2 this means that when the voltage between the gate and the source is smaller than a certain threshold, the

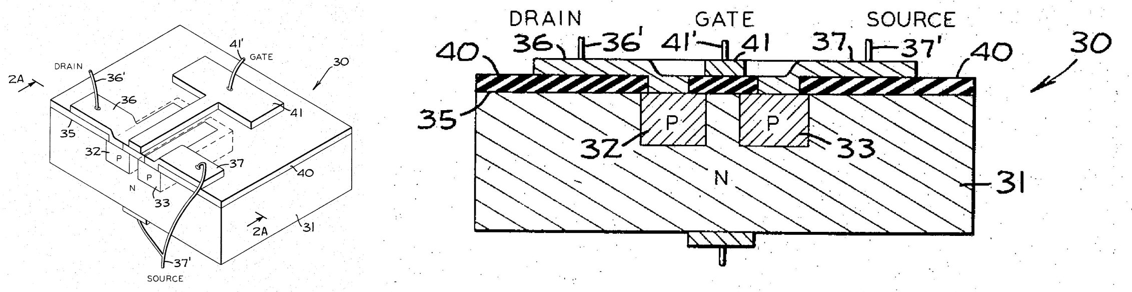


Figure 1: Layout drawing of a PMOS transistor as seen in the patent US3356858A by Wanlass (1967). The left side shows an overview of the wafer, while the right side shows the cross section.

current can flow from drain to source. If the potential difference increases, the current flowing through the transistor will increase accordingly.

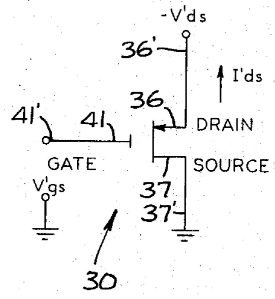


Figure 2: Schematic drawing of a PMOS transistor as seen in the patent US3356858A by Wanlass (1967) with the terminals wired to the according voltage sources.

Looking at an NMOS transistor, the doping of the substrate and therefore the doping of the connections is switched, see figure 3, resulting in an opposite reaction. For gate-source voltages below a threshold, the current from drain to source is basically zero. When increasing the potential difference, the resulting current also increases.

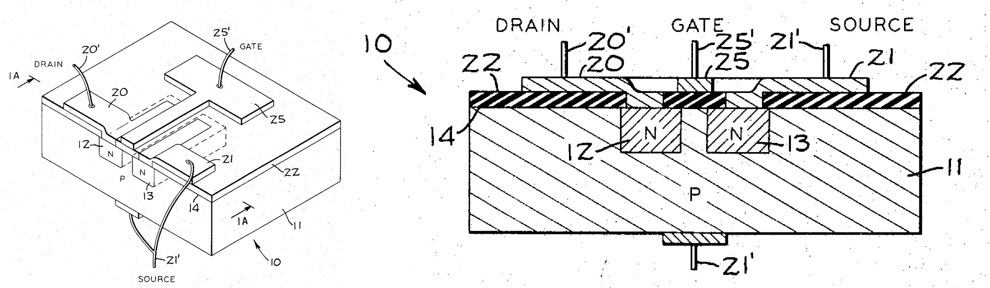


Figure 3: Layout drawing of an NMOS transistor as seen in the patent US3356858A by Wanlass (1967). The left side shows an overview of the wafer, while the right side shows the cross section.

For deeper explanations of the different functions and operation options of the transistors one can refer to different kinds of standard electronic teaching materials. Suggested readings that have already been mentioned would be (Razavi, 2013) and (Phillip E. Allen, 1987) as well as (Jaeger and Blalock, 2011). After briefly introducing the two different MOS transistors, a short explanation, why it can be beneficial to have PMOS and NMOS transistors on the same silicon wafer, will be given. One use case utilizing both transistor types is the inverter. Here an NMOS and a PMOS are wired in parallel to an input, with the PMOS drain connecting to VDD and the NMOS source connecting to GND. This configuration essentially creates an output of VDD, if the input is set to GND and vice versa.

However, creating both of these transistor types in the same silicon is not as easy as it may sound. When looking at the doping type of the substrate, it's clearly visible that the PMOS transistor is sitting in n-doped silicon, while the NMOS transistor is sitting in a p-doped substrate. The idea of CMOS is to combine these substrates by creating a p-well inside the n-doped substrate, which allows placing the NMOS next to the PMOS. In most common processes used today the doping is reversed, resulting in the creation of an n-well inside a p-doped silicon wafer. But whatever the difference in doping, today's processes are based on the patent presented by Wanlass (1967) introducing the CMOS technology. A sketch of his idea is presented in figure 4.

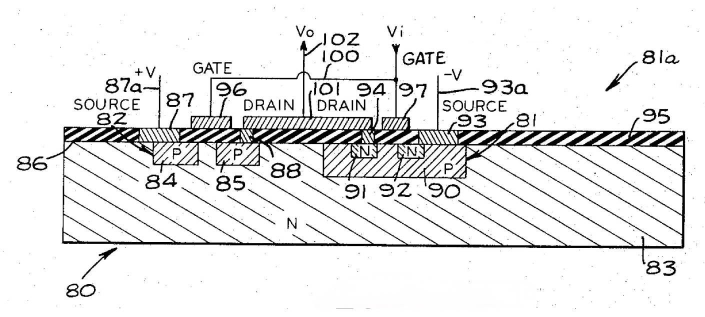


Figure 4: Cross section of a CMOS layout with an NMOS and a PMOS on the same wafer, as seen in the patent US3356858A by Wanlass (1967). The shown configuration represents an inverter.

2.2 The BrainScaleS-2 System

As mentioned in the introduction, this work involves the creation of a scalable synapse driver array for use in the BrainScaleS-2 (BSS-2) neuromorphic system. The term neuromorphic describes hardware used to mimic nerve structures, especially those of the human brain (Billaudelle, 2017). One of these neuromorphic systems is the BSS-2 system created and maintained by the Electronic Vision(s) group at the University of Heidelberg. The research group introduced their first mixed-signal neuromorphic chip, Spikey, which was based on a 180nm CMOS technology, with the work by Schemmel et al. (2006). Back then the system contained 384 neurons and 256 synapses (Pfeil et al., 2013). The latest system is the already mentioned BSS-2 system, which is manufactured in a 65nm process, containing 512 neurons with 256 synapses per neuron (Pehle et al., 2022). It is based on an analog model of physically implemented neurons and synapses that, when combined, result in an analog neuromorphic accelerator which uses a continuous-time to emulate the spike-based dynamics of a neural network. Interestingly, while the number of synapses and neurons falls short compared to the real world example, the time scales in this system are up to a thousand times shorter than the real world comparison, resulting in a high acceleration in time compared to the biological counterparts (Schemmel et al., 2020). A single BrainScaleS-2 core includes the full custom analog part holding a synaptic crossbar combined with neuron circuits, analog parameter storage, two digital control- and plasticity-processors, and an event routing network responsible for spike communication. A schematic floorplan is shown in figure 5. Here you can see that the design was split into four quadrants containing 256 by 128 synapses each.

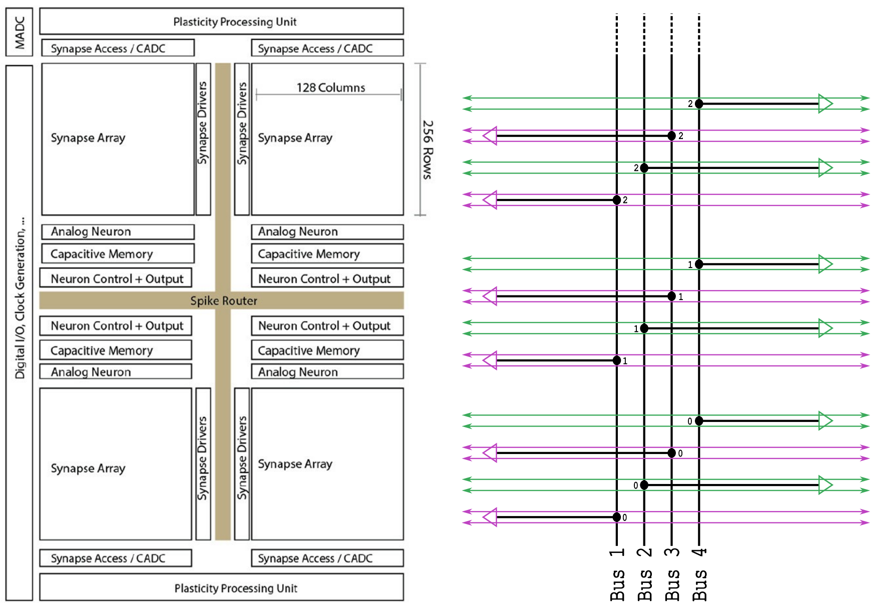


Figure 5: The left side shows a schematic floorplan of the BrainScaleS-2 system as described in the work of Pehle et al. (2022), while the right side shows a representation of a synapse driver schematic. The bus lines are depicted in black, connecting the drivers shown as triangles. The driver outputs as well as the drivers are colored green and pink depending on the position (left or right) of the drivers. The right picture can be understood as zoomed in on the synapse driver on the left.

This floorplan further shows synapse drivers, which are used as the interface between the event routing bus and the analog core of the chip. In other words, the drivers take the inputs from the digital part and feed them into the analog part of the chip. Here the digital event handling logic is used to inject these events into different decoders, acting as a custom CMOS-level bus and distributing the spike events across the array of synapse drivers. To achieve this, the synapse drivers derive timing signals for the synapse circuits and drive them across the synaptic rows using a 6-bit synapse address as well as an enable signal (Grübl et al., 2020). Four of these 6-bit busses are used to activate the corresponding synapse drivers. Here each synapse driver manages two synaptic rows per side resulting in four synapse rows per driver (Billaudelle, 2017, p. 12). On focusing only on the top half of the chip, one can draw a simplified connection plan of the synapse drivers as shown in figure 5. This is possible since the lower half of the chip is essentially a mirror image of the upper half.

Furthermore, the synapse drivers also implement the short-term plasticity model (STP), following the pre-synaptic implementation approach as tested in the previous chip generations (Grübl et al., 2020). However, with the aim of this work in mind, it can be concluded that the detailed implementation of this model is not relevant. Therefore no further information about the STP implementation is given.

A closer look at the floorplan of the BSS-2 system shows that the system is quite complex, and therefore also quite space consuming, resulting in the need for a large die-area. This is not a problem in itself, but it does mean that each individual chip, especially when considering smaller improvements from one generation to the next, is quite expensive to manufacture. One of the reasons is the direct relation between cost and chip area used. While these large die-area chips are indispensable in some instances, for example when scaling up systems, most of the time it is possible to test changes of smaller components individually. When testing the individual components, the production of small scale chips or so called mini asics, keeping costs for each iteration at a minimum, would be optimal. Coming back to the BrainScaleS-2 system, the question on how to achieve these small scale chips arises. Here the idea was to find a way how the design could be shrunken down for testing purposes. Based on this, the objective of the work set out in this thesis was to be able to scale down the system in number of synapses and neurons. To achieve this, the first three main components that have to become scalable in design are the synapse driver array, the synapse array and the neuron array. Out of these components the synapse driver array is dealing with different connections the most. This is the reason why it was chosen to serve as an example and possibility check for creating a scalable neuromorphic design using PCells.

2.3 VLSI design flow

To understand what is meant by the term PCell and why it is desirable to use them in circuit design one first has to understand the design process behind VLSI systems. The term Very-Large-Scale Integrated Circuit (VLSI) is used to describe the ever growing integrated circuits (ICs) on a single silicon wafer. These integrated circuits or ICs describe the realization of a large number of components, mainly transistors, on one wafer with the main benefit that the devices share quite similar characteristics due to the equal manufacturing conditions. This ability to create devices with nearly matching characteristics has resulted in the development of special circuit techniques taking advantage of these similarities (Jaeger and Blalock, 2011, pp. 1046–1049).

The design process of these integrated circuits is based on many different steps. While there is no perfect design flow used by everyone, the key steps of these flows will still be quite similar. To showcase a possible design flow, an introduction into 5 crucial steps will be given. The naming of these steps can be subject to changes depending on the process in question. In terms of naming and step order, this work is based on the workflow presented by Cadence®(Cadence Design Systems, 2023b) as a result of the later use of their design tools.

Following these recommended design steps, all processes should start with an "architectural design" part. This means thinking about what exactly is to be achieved. Furthermore, all the organizational steps, such as the expected costs and use cases of the IC, are taken into account. After the requirements as well as the boundaries regarding a specific IC are fixed, the "logic design" begins. Here one takes the requirements from the "architectural design" and tries to break them into increasingly smaller blocks. This helps to create a hierarchical idea of the aspired design. When all small building blocks are determined, either preexisting similar blocks are used and modified when possible, or new ones, solving the design challenges are created. This process mainly works on the schematics of a design. After finishing the schematics of the desired IC, it is advised to perform the first simulation tests verifying that the idea and function of the desired circuit have been achieved. If these tests prove successful, the next step is to create the "physical design" of the IC. In this step, all the small building blocks from the schematics have to be implemented in the layout of a chip. Here one has to determine the location, size and shape of all the different modules, effectively drawing the masks used for producing the wafers. It is advisable to re-verify each small building block against its schematic counterpart to ensure that no mistakes are made. With increasingly higher hierarchical levels it also becomes increasingly difficult to find and extract errors.

Combining this "physical design" step with the hierarchical design of the chip also provides the first strong argument for creating scalable and reusable design parts. One option to achieve this would be to copy the small design parts each time they shall be used and then modify them according to the specific needs. But as one can imagine, over a longer period this becomes space and time consuming, being the reason why this option is not really desirable. Another reason why this option is not optimal is the fact that the results are not scalable without a considerable amount of work. For example, if one wanted to double the size of an old design, one would have to double everything by hand which is a quite time consuming process. One option to resolve these issues are so called PCells, or Parameterized Cells, which can be created using Cadence® Virtuoso®. These can then be used as the small building blocks of the design. Here it is also possible to incorporate these small parametrizable cells into bigger Parameterized Cells, which finally can result in a fully scalable and modifiable IC design.

Coming back to the last two steps of the design process, the step after completing the "physical design" of the IC is to verify the created designs. This process is also called "physical verification". Here the physical design on the masks for the wafer is primarily tested both for design rule violations, assuring that the design is manufacturable, as well as for differences between the layout and the schematics from step two. If these tests are successful, simulations of the actual layout, taking into account real life effects like crosstalk and resistances, are carried out. Most of the time the design is also tested at the edges of the manufacturing variations, thereby checking the stability of the design. With all the desired tests passed, the final step of the process is the "signoff". In this step the most critical parameters are verified again, including timings, power consumptions and signal strengths. If all necessary steps are taken, the circuit will be sent to a manufacturer, where there will be some final testing, followed by the manufacturing of the wafers. This design flow is also modeled in a flowchart shown in figure 6.

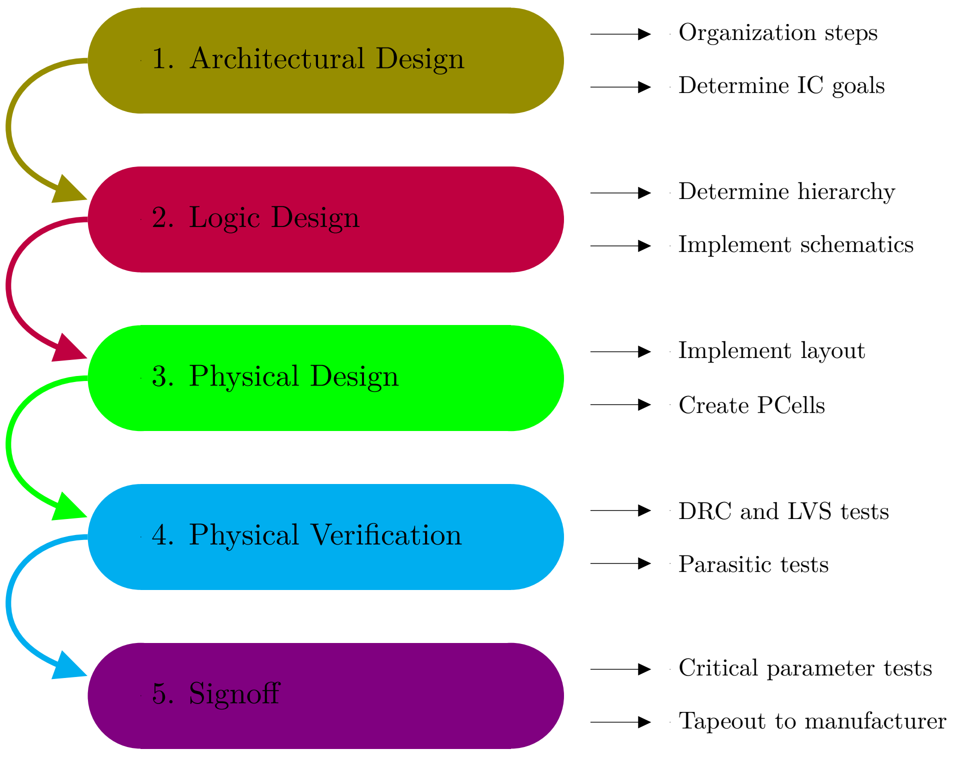


Figure 6: Design flow chart for ICs inspired by the explanation from Cadence Design Systems (2023b) and the work by Patni (2021). The design flow chart shows the different steps of the process in the colored boxes with short explanations of these steps on the right.

2.4 PCells

As motivated in the previous paragraphs it can be quite desirable to implement scalable and reusable designs when creating ICs. Here the PCell from Cadence® Virtuoso®, as mentioned above, is a powerful tool to achieve this flexibility. In short, a PCell is a layout, schematic or symbol of a cell used in Virtuoso®. The distinctive feature of these cells is that they have parameters attached to them, allowing for customized appearances when inserted into a design. One of the smallest examples for a PCell would be a transistor, most of the times supplied as part of the process design kit belonging to a CMOS technology. Here some cell parameters would include the length and the width of the gate, as well as for example the number of fingers or multipliers a transistor holds. When changing these parameters for instances inside a design, the model of the transistor changes accordingly. The same holds for the use of the PCell in a layout. Here a parameter change results in physical change of the drawn transistor. Using the example of the transistor one can easily show that it is possible to use the same basic design, in this case the transistor, in many different implementations. Cadence® allows the user to create such PCells on their own, thereby providing quite a powerful tool for the purpose of IC design.

Working with these PCells, Virtuoso® provides two options on how to create them. One can either create the PCells graphically, using the options inside the PCell menu, or textually, using the programming language of Cadence®, SKILL (Cadence Design Systems, 2004). While it might be appealing to create the PCell graphically, the complexity of the PCells created that way is quite limited. When creating the PCells using SKILL, the options, and therefore the possible complexity of these PCells, are nearly unlimited. This is a direct result of the fact that everything the design software Virtuoso® is capable of can also be achieved using the correct SKILL commands. This for example is reflected in the parameters, which can be used quite freely, resulting in very little to no boundaries when creating PCells for own designs. Because of the greater flexibility, and the option to create schematic and symbol PCells, the method used to create the cells shown in this work is using SKILL.

In addition to scalability and reusability, PCells carry the following advantages when used in design processes. They are able to save disk space by using only paths back to the original cell instead of saving all cells individually, they speed up the layout data by eliminating the need for duplicated parts, their designs can be maintained way easier because all changes can be done in the parent cell and finally, they allow for much easier changes in the designs by using parameters instead of having to go through a stack of layout layers (Cadence Design Systems, 2004).

2.5 Simulation and parasitic extraction

When looking at the design flow of ICs as explained above, it is visible that in addition to actually creating the circuits, a big part of designing is simulating and testing the different designs at varying design stages. When working with schematics, this can be achieved by creating a so called netlist of the drawn circuits. These netlists are used for simulating the designs by applying different component models or calculating the different behaviors according to basic electronic equations. Most of the time the software used to generate these simulations is based on SPICE (Simulation Program with Integrated Circuit Emphasis). SPICE in its core is an open source circuit describing language used for simulations of complex circuits by replacing the used components with different models. When simulating the circuit all components, as well as their connections, get translated into a netlist which itself is used to generate equations describing the circuit. These detailed equations are mostly solved numerically, with a precision that is not achievable by hand in a reasonable time (Jaeger and Blalock, 2011, p. 167). To get an understanding of the complexity of these component models the level-1 SPICE model of an NMOS is represented in figure 7.

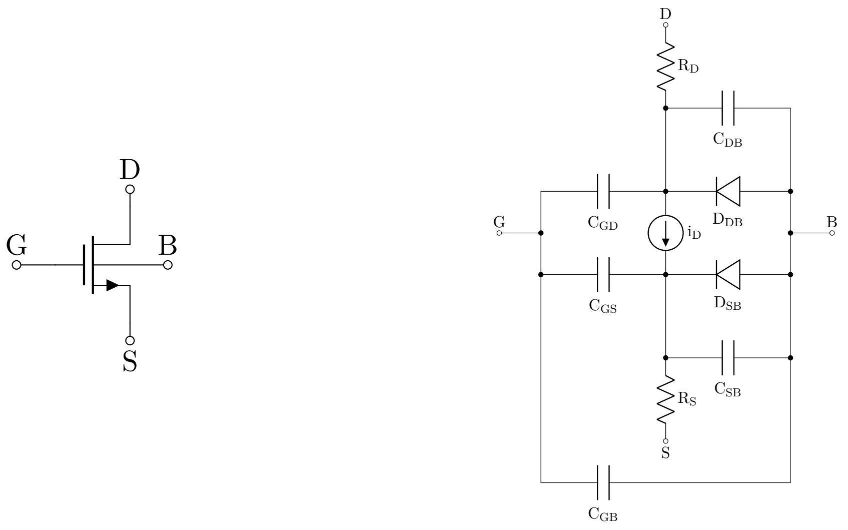


Figure 7: The left side shows the symbol of an NMOS transistor as used in modern processes. The right side shows the Level 1 SPICE representation of this NMOS as seen in the book by Jaeger and Blalock (2011, p. 167) or Phillip E. Allen (1987, p. 104)

Here it becomes clear that even for a "simple" MOSFET the number of equations describing the whole system can become a nuisance.

In addition, these netlists used for simulations can be extracted from the layout using the GDSII data and for example the tool Calibre. Using this functionality it is now possible to compare these two netlits of layout and schematic against each other. The process of comparing these two is also known as layout versus schematics (LVS). Here it is possible to spot differences or errors in the designs, which may have been missed when creating the layout. Naturally, if it is possible to create a netlist from the layout, it is also possible to use this netlist for simulations. Interestingly the simulation options for the layout exceed the ones of the schematics. The reason for this are the so called parasitic extractions of the layout generating a netlist not only containing the connections, but also the physical effects, like crosstalk or line capacitance, that the shapes drawn for the wafer masks will likely have on each other. These extractions are needed, because the extended use of lower resistive conductors on chips combined with the increasingly tight on chip timing requirements, resulted in an increased visibility of unwanted effects (Kao et al., 2001). To calculate these parasitic effects different methods of extracting can be used. Providing a better understanding of these extractions, some of the widely used models will be introduced in the following paragraphs.

When talking about these parasitic extractions the main contributors are the resistance, the capacitance and the inductance among the interconnected wires. Of these, the calculation of the capacitance is playing the most significant role. To extract these parasitic capacitances, two main processes, the field solver method and the pattern-matching method (Ma et al., 2023), are used. The field solver works by directly stimulating the electrostatic field and saving the results, therefore achieving highest accuracy. As always, this accuracy comes with the penalty of quickly rising computing times as well as massive memory consumption. It therefore is only suitable for smaller designs or isolated design parts (Yu, Song, and Yang, 2021).

The method based on pattern-matching is the one used for working with circuits of increasing size or even full chip extractions. It also is the foundation of most industry adopted extraction tools like Calibre by Siemens. This method works by combining three steps of extraction, of which only one is executed on a per design basis. These three steps are: first the generation of different wire patterns covering a wide range of possible designs, second the calculation of capacitance models for these structures, and third an extraction of the actual circuit layout, from which the found wire structures are mapped to the pre calculated models. The advantage of these pre calculated models is that they only have to be calculated once for each process technology and therefore are independent of the actual design. This means that they are supplied with the rest of the technology files. These pattern capacitance libraries are created using a field solver, as mentioned above, and can consist of several thousand structure patterns, see the work of Yu, Song, and Yang (2021).

Interestingly there are different models for capacitance field solvers, mainly differing in the dimensional complexity of the extractions. During the ongoing developments of these extraction models, they evolved from taking into account the 1-D effects, over 2-D and 2.5-D to considering 3-D effects. Each of these model revisions surpassed the lower dimension ones in the accomplished accuracy. While the 1-D extraction only accounted for the area and perimeter of interconnected geometries, combined with an average environment of these wires, the 2-D extraction included capacitive effects in relations between wires of the same lateral metal layer. Because it is not as trivial as it seems to extend these 2-D extractors into the third dimension, a 2.5-D solver has been developed (Kao et al., 2001). The 2.5-D extraction works by orthogonally combining two 2-D extractions. To achieve this, the layout is first extracted horizontally and then vertically. Combining these two extractions allows for the modeling of most 3-D effects. Examples of how these two views might look like are shown in figure 8.

Nowadays mostly 3-D solvers are used for the extractions. While there are the traditional deterministic methods, like the boundary element method (BEM), or the finite difference method (FDM), numerically solving the maxwell equations and describing all the different metal layers in three dimensions (SimTech, 2023), current research tries to reduce the computation time and power as well as the memory space needed for these extractions. One attempt on completing this task is the implementation of a floating random walk to generate the dependencies, as shown in Yu, Song, and Yang (2021). This work shows

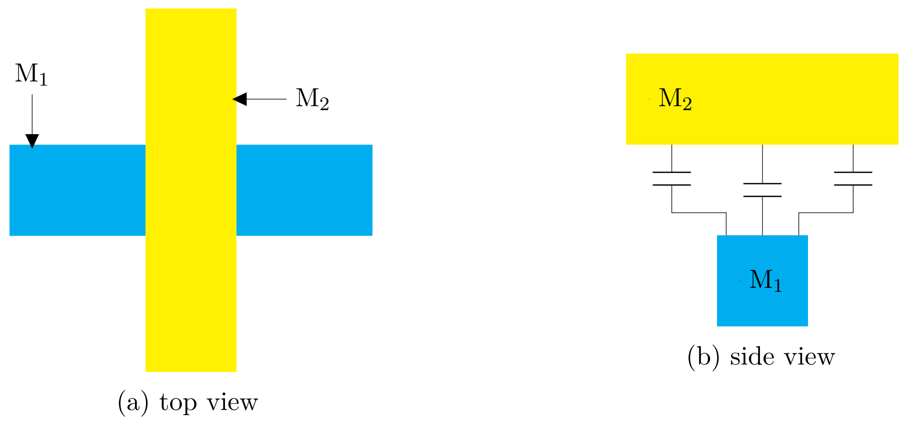


Figure 8: Schematic depiction of 2.5-D modeling for two metal lines in different layers. Inspired by the work from Kao et al., 2001

the importance of speeding up the extraction and simulation process, without having to reduce the complexity of the designs. Moreover, this gives a foundation for the second goal of this work, trying to decrease the simulation times needed for big repetitive designs. This shall be achieved by configuring the synapse driver array as a PCell in a way that it becomes possible to replace most of the repeating design parts with extracted models, essentially speeding up the simulations, while also keeping the accuracy of the simulations at a reasonable error.

3 Methods

In the process of creating and simulating the PCell of the synapse driver array for the BrainScaleS-2 system different tools have been used. The following few paragraphs are going to give a short introduction explaining the tools used as well as their purpose.

3.1 Cadence® Virtuoso®

Most of the work regarding this thesis was done via the Cadence® Virtuoso® software. Virtuoso® is mainly used to create custom IC designs in schematic and layout. The software also includes a tool for analysis and circuit simulation. The Virtuoso® studio software is the world leader regarding circuit and chip design. Regarding the software Cadence® is cooperating with chip manufacturing companies like TSMC and UMC. The simulation solution, Spectre, which is integrated into Virtuoso® is capable of performing both analog and digital simulations, as well as mixed signal simulations. While it is based on SPICE, Cadence® claims that the simulation speed is up to ten times faster than ordinary SPICE simulations (Cadence Design Systems, 2023a).

3.2 Calibre

The Calibre design solutions portfolio is a piece of software that is maintained by Siemens and can be used in the IC sign off verification and optimization process. In the context of this work Calibre was used to perform the layout versus schematic (LVS) and Design Rule Checks (DRC), as well as to extract the parasitics of the designs using the parasitic extraction (PEX) tool. The checks were executed using the allowed integration of third party software into the Virtuoso® design suite by running the Calibre verification tools directly from the cellviews graphic interface.

3.3 SKILL

As mentioned before, the PCells in this work have been created using the programming language SKILL. SKILL as a programming language is a dialect based on a dynamic scoped Lisp interpreter. It was developed to be used as an extension language for CAD systems with the goal of providing tools and design control for the user on a level unachievable otherwise. Because it was developed with the user in mind, the actual parser is quite less restrictive than usual Lisp. This simplified parser for example allows a far more C-like syntax (Barnes, 1990). Nowadays SKILL is tightly integrated into the Virtuoso® design suite, allowing for full control over the designs and software by only using the command line.

4 Creation of synapse driver and address decoder PCells

After introducing the main topics of this thesis, the following chapters will deal with the parametrization process as well as the challenges and solutions that arose during the process of creating a parametrized synapse driver array. Here the different steps and decisions will be discussed, while also giving code examples for implemented solutions. The work regarding the scalability of the synapse driver was started by taking a dive into the design blocks of the fixed array, used in the latest BSS-2 system.

4.1 Fixed synapse driver array

The old synapse driver array was designed and constructed following the traditional design flow as explained before. This means that it was constructed out of many different layers of instances, used in an ever growing hierarchy. Specifically, this means that all the different parts of the design, such as the bus, the drivers or the SRAM decoder were all created and placed by hand. Further this layout was extended using a custom SKILL script creating the decoding via patterns on top of the instances, as well as using a second custom SKILL script, creating the labels used for net naming and simulation on top of the layout. In some places, where the features of the array have been expanded over time, metal shapes, used for those features, were drawn right on top. Starting from this, the first task in changing this array into a scalable PCell was trying to replace the via script, creating the decoding pattern in the incoming bus lines, again see figure 5, with a changeable decoder that, making use of its flexibility, hopefully will be used in many different upcoming designs. The process of creating this decoder, ultimately achieving the desired functionality, is highlighted in the following paragraphs.

4.2 Address decoder

Combining the argument of trying to get rid of custom built solutions for each new design with the almost infinite possibilities provided when creating PCells, the first decision that had to be made during the creation process of this changeable decoder design was the parameter selection. Here the parameters one would likely use in the future as well as their implementation had to be chosen.

4.2.1 Parameters

Addressbits

Deciding on the parameters, the first one to be implemented was the number of address bits for the input bus. This parameter takes the form of an integer, allowing to choose any whole number larger than zero. Implementing this parameter was fairly straightforward, because the usage of an integer meant that it could just be passed into the design. Setting this parameter defines the number of address lines, hence the naming, while in the same step also defining the number of routing outputs.

Custom Length and Maximum Bit

To accompany this parameter two further parameters allowing for a smaller number of outputs have been added. The reason for this was that sometimes one has an input of a certain number "a" of address bits, but does not need all 2^a outputs. If this parameter is not used, the number of outputs is determined automatically. The first parameter added allows a reduction in output numbers and therefore is called custom length. It is implemented as a boolean with the default `false (nil)`. When set to `true (t)`, the second parameter is revealed. This normally hidden parameter is called maximum bit and is also implemented as an integer. When used with a value smaller than 2^a, the PCell only calculates, and therefore creates, routing outputs up to the specified number.

During the process of creating these two parameters two challenges arose. The first one was faced while creating parameters that normally are hidden and uneditable, but turn into "real" parameters, depending on the values of other parameters. Here the first idea was to use a callback attached to the custom length parameter, changing the value of the display option from the maximum bit parameter. Unfortunately, that did not work, which is the reason why the current implementation is making use of the `?display` option directly attached to the maximum bit parameter. The code used to implement this functionality is shown below.

```
cdfCreateParam( cdfId
    ?name "maxBit"
    ?type "int"
    ?prompt "Maximum Bit"
    ?defValue 2**8
    ?display "cdfgData~>customlength~>value == t"
    ?editable "cdfgData~>customlength~>value == t"
)
```

Code 1: Code used for defining the `Maximum Bit` parameter with the `display` and `editable` option.

The second problem that had to be solved was that in SKILL `nil` is used for the boolean `false` as well as for undefined values. This resulted in a rather strange behavior where, if the default of a parameter is set to `nil`, the parameter still gets created and is accessible, but unfortunately, the changing of the boolean value is not registered internally. To derive the type of the parameter, a statement defining which `nil` was meant, by stating a `"boolean"` before using `nil`, has been implemented. Looking at the decoder code, this can, for example, be seen in the parameter defining part at the beginning of each view type. (Code 6)

Decoding and Custom Decoder List

Following the parameter creation, the next parameter implemented is the decoding. Here one can choose between a "standard" backward or forward decoding as shown for example in figure 19. Further a custom decoding option was added, which stays hidden, using the same method as the parameter code shown above in Code 1. In this custom decoding one can enter an arbitrary decoding represented by a list. An example of this, containing 3 address bits, would be `(0 2 4 8 1 3 5 7)`, resulting in a separation of even and odd addresses. In the actual code this is implemented by calculating the bitwise representation of each routing output, following the given list, and then using an `if` query to decide which input the output should be connected to. This will be explained in more detail in the layout section later on. Looking only at the schematic and symbol of the decoder these parameters are the ones actually used inside the PCell. However, for creating the layout of the decoder further parameters have been created.

Routing Layer, Address Layer and Inverse Address Layer

One big flexibility the design has to account for in the layout are the different metal layers one can and has to choose from when drawing the conducting metal lines. To achieve this flexibility three further parameters were added into the design. These parameters allow the user to choose between different metal options for the routing layer, as well as for the address and inverse address layer individually. Here one really nice design feature is that the options presented to the user change depending on prior choices. In explicit terms this means that when for example "metal 3" is chosen for the routing layer, the address and inverse address lines only can be drawn on the layers "metal 2" or "metal 4". One of the reasons why this restriction was implemented is that in most use cases the broader option overcomplicates things by allowing the user to create vias that can connect through more than one hierarchical metal layer at once. Keeping this in mind the restriction can provide a bit of support for the user implementing this decoder into a new design. To achieve these dynamic choices, a callback in the definition of these parameters is used. From there the options of the other two parameters are altered by creating a list, depending on the own settings. An example of this code, in this case showing the callback from the routing layer parameter, looks as follows.

```
cdfCreateParam(cdfId
    ?name "routingLayer"
    ?type "cyclic"
    ?prompt "Routing Layer"
    ?defValue ""
    ?choices list("" "M1" "M2" "M3" "M4" "M5" "M6" "M7" "M8" "M9")
    ?callback "
    layeroptions = list(\"\")
    layeroptions = append(layeroptions list(strcat(\"M\" sprintf(nil \"%d\"
    ↪  atoi(substring(cdfgData~>routingLayer~>value 2))-1))))
    layeroptions = append(layeroptions list(strcat(\"M\" sprintf(nil \"%d\"
    ↪  atoi(substring(cdfgData~>routingLayer~>value 2))+1))))
    cdfgData~>addressLayer~>value = \"\"
    cdfgData~>addressLayer~>choices = layeroptions
    cdfgData~>invaddressLayer~>value = \"\"
    cdfgData~>invaddressLayer~>choices = layeroptions
    "
)
```

Code 2: Code used for defining the `Routing Layer` parameter with the `callback` option used to change the values of the `Address Layer` and `Inverse Address Layer` parameters.

Metalwidth

Another parameter used only inside the layout of the decoder is the width of the drawn metal lines. Here the parameter is implemented as a string, allowing the user to input values like for example "0.18u". This parameter is used for the width of the address lines, as well as for the routing lines. Here it would be possible to argue that it may be beneficial to add a separate width for each type of metal, nevertheless, the decision was taken to use only one metal width allowing easier via placement, as well as creating a simpler interface. The other thing that automatically changes with the metal width is the position and the number of placed vias. This enables a good connection between the address and routing lines by increasing the number of placed vias according to the cross section.

Address Position, Inverse Address Position, Routing Position and Routing Pitch
Another important parameter added for creating the metal shapes is the position of the lines in respect to each other. To achieve a flexibility in the changeable pitch between the lines, four parameters, all implemented as a string, are used. The four parameters that are used are called "Address Position", "Inverse Address Position", "Routing Position", and "Routing Pitch". The first three are used to calculate the exact positions on which the respective metal lines will be centered, while the routing pitch simply is the distance from the first row of bundled routing outputs to the second row.

In each of these parameters, the user has to specify a simple arithmetic expression. Fulfilling the purpose of generating the positions, the given strings are evaluated making use of increasing variables, for the address line "b", and for the routing line "i". This allows the user to create a wide range of arrangements of pitches, for example by using if statements for the individual positions, while also maintaining a quite simple interface for creating "standard" pitches. Examples for these possible pitches are shown in figure 9.

Creating these string interpreters, again different challenges had to be overcome. The biggest of these challenges was trying to prevent the possibility for malicious attacks exploiting the strings when evaluated with an `evalstring()`. When only executing an `evalstring()` on the parameter string, it is possible to gain full access to the underlying system by exploiting the parameter textbox as a command line. To create a defence against the possibility of exploiting this function, an implementation of the Automate Expression language (AEL) interpreter from SKILL has been tested.

However, the limited application potential combined with the compatibility problems between the AEL functions and PCells (Cadence Design Systems, 2004), resulted in the decision to not use the AEL interpreter. Problematically, this again led back to the use of a simple `evalstring()`. To solve this problem, a regular expression checker was added to the code. Now it is possible to expand the allowed functions or symbols in the parameter as needed, while still assuring that an attack exploiting this `evalstring()` is quite unlikely. The current implementation works by checking the string with a custom SKILL function given below.

```
procedure( allowedString(input)
    let((var res)
    rexCompile("mod")
    var = rexReplace(input "000" 0)
    rexCompile("if")
    var = rexReplace(var "00" 0)
    rexCompile("then")
    var = rexReplace(var "0000" 0)
    rexCompile("nth")
    var = rexReplace(var "000" 0)
    rexCompile("[^0-9\\*\\.\\+\\/\\ \\(\\)ibunp\\<\\>\\=\\-]\'")
    if( rexExecute(var) == nil then
        res = t
    else
        res = nil
    )
    res
    )
)
```

Code 3: Code defining a function that is used to check an inserted string for words and characters that are allowed for use in the definition of the layer positions. The function returns `t` or `nil`, depending on the outcome of the check.

This code works by taking a given string and mapping the allowed functions exactly onto the string. Then these allowed functions are replaced with "`0`"s, ultimately removing the allowed functions from checking. This is achieved by making use of the `rexReplace()` function. Once the allowed functions are replaced, the string is checked for the allowed singular characters. In the code example above the allowed singular characters, containing mathematical symbols such as `*` or `+`, as well as some single letters used as units, like `u` and `n`, or as the variables `b` and `i`, can be seen. Further the few allowed functions `mod()`, `if()`, `then()` and `nth()` can be found.

If the string inserted to the parameter does not contain any illicit characters or functions, the check function returns a boolean `t`. Otherwise it returns `nil`. The function checking the inserted code is called with a callback attached to the parameter resulting in an execution each time the value is changed and most importantly before the string is passed into the different views for evaluation. If `nil` is returned, the parameter will not be forwarded and instead an error will be presented inside the text field for the user. The code used in the callback looks like this.

```
?callback "
    if(allowedString(cdfgData~>routingSpace~>value) == nil then
        cdfgData~>routingSpace~>value = \"not allowed function detected\"
    )
"
```

Code 4: Code defining the callback of the parameters used for metal positions. The `allowedString()` function shown in Code 3 is executed and the string is parsed according to the return.

To further present the capabilities of this string interpreter some functions generating different metal arrangements will be shown. This showcase will include some simple functions like `b*1u+0.5u` for address placements, as well as some rather complex functions like `nth(i '(0 1 2 3 4 10 9 8))*0.3u` for the routing lines. The functions used for this showcase, as well as the created layout placements, can be seen in figure 9. Here all metal widths stay fixed at 0.24µm.

Address Lines, Draw, and Inverse Address Lines, Draw

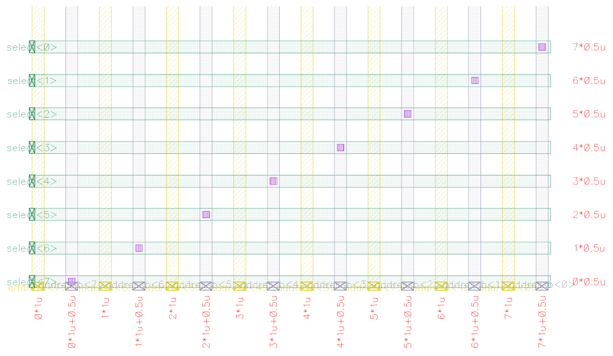
Allowing a reproduction of the via script previously used, the option to turn off the drawing of the metal lines has been added. Here the user can deactivate the drawing of the address, inverse address and routing lines individually, ultimately creating a cell that only places the vias at the positions where the metal lines would cross. At first glance this might seem like an unnecessary option to add, but when creating cells with repeating designs it can be the case that the metal lines of the routing are already drawn in the cell of a lower hierarchy. If this is the case, the metal lines already are part of the design and do not have to be redrawn in this higher hierarchy.

Help

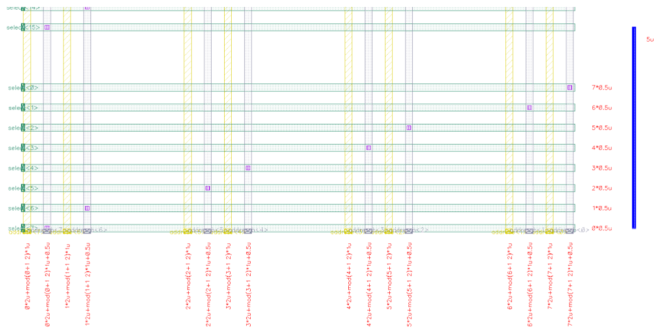
To guide the user through the process of setting up the via matrix, especially in the light of the more advanced features, a help button was added as a further parameter. When pressed, this button opens a file containing an explanation for each parameter, as well as some examples on how to use them. To achieve this functionality, the file containing these explanations has to be accessible to every user that implemented the code in their library. To ensure this, a way to create the file directly through SKILL was implemented. Here the functions `outport1 = outfile("path")`, `fprintf(outport1 "content")`, `drain(outport1)` and `close(outport1)` have been used.

Combining all these parameters, the input mask created for this PCell can be seen in figure 10.

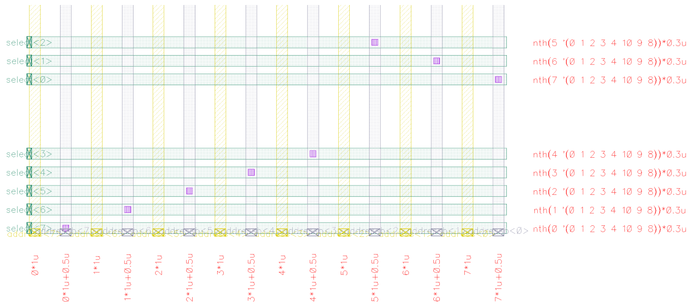
This input mask concludes the main parameter creation process and also provides an overview of the parameters and functionalities created. While there might be special cases in which the parameters created here may not be sufficient enough, the further course of



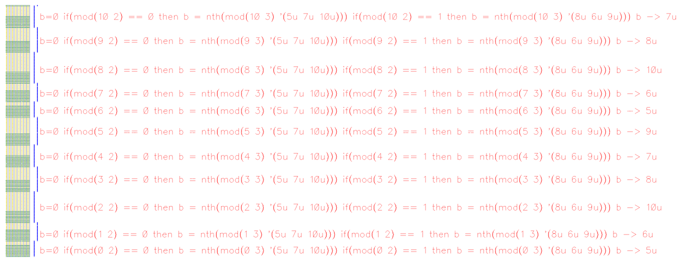
(a) Address Position: b · 1u, Inverse Address Position: b · 1u + 0.5u,
Routing Position: i · 0.5u, Routing Pitch: 5u



(b) Address Position: b·2u+mod(b+1 2)·1u, Inverse Address Position: b·2u+mod(b+1 2)·1u+0.5u,
Routing Position: i · 0.5u, Routing Pitch: 5u

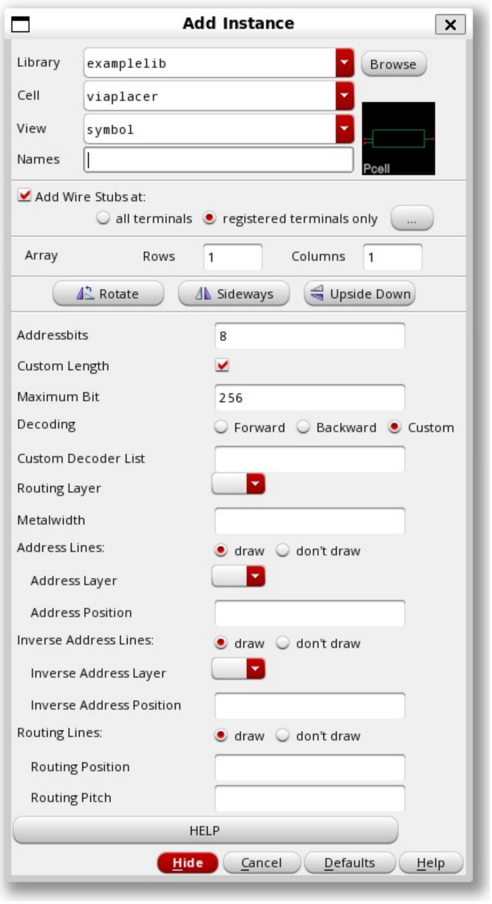


(c) Address Position: b · 1u, Inverse Address Position: b · 1u + 0.5u,
Routing Position: nth(i '(0 1 2 3 4 10 9 8)) · 0.3u, Routing Pitch: 5u



(d) Address Position: b · 1u, Inverse Address Position: b · 1u + 0.5u, Routing Position: i · 0.5u,
Routing Pitch: b = 0 if(mod(i 2) == 0 then b = nth(mod(i 3) '(5u 7u 10u))) if(mod(i 2) == 1
then b = nth(mod(i 3) '(8u 6u 9u))) b

Figure 9: The layout of the decoder PCell showcasing some different possibilities in creating and placing the metal lines. The parameters used for generating the depicted layouts can be seen in each shown subfigure.


Add Instance

Library	examplelib
Cell	viaplacer
View	symbol
Names	

✓ Add Wire Stubs at: all terminals [✓ registered terminals only]

Array	Rows 1	Columns 1	

Addressbits	8
Custom Length	✓
Maximum Bit	256
Decoding	Forward Backward [✓ Custom]
Custom Decoder List	
Routing Layer	
Metalwidth	
Address Lines:	[✓ draw] don't draw
Address Layer	
Address Position	
Inverse Address Lines:	[✓ draw] don't draw
Inverse Address Layer	
Inverse Address Position	
Routing Lines:	[✓ draw] don't draw
Routing Position	
Routing Pitch	

Figure 10: Input mask of the address decoder PCell that is opened when the decoder is inserted into a design in Cadence® Virtuoso®.

this work will hopefully show that using the PCell parameters as implemented all options the further work requires are sufficiently covered. Before showcasing the use of the decoder inside the synapse driver array, the creation of the decoder cellviews with their respective problems, as well as some tests and simulations, will be shown.

4.2.2 Cellviews

Layout

Most of the problems that occurred during the layout creation of the decoder PCell had to do with calculating the right positions of the metal lines and placing the vias. Concerning the size of the vias it is important to stick to the design rules, on the one hand, regulating the size and spacing of the vias, while on the other hand trying to maximize the possible contact area between the metal layers. To achieve this, an `if` query was implemented, handling all metal widths below the size of 0.2 µm. For all vias that have to be placed on wiring with a metal size wider than 0.2 µm, the positions are calculated by a create vias function, that given a centerpoint and area fills the given area with as many vias as allowed by the design rules. This function can be found in the file defining all functions used in different code parts, as well as in Code 8.

Another challenge was the creation of the decoding inside the layout. To solve this challenge an approach using bit conversion as mentioned before was implemented. Here the decoding, given as a list of decimals, is iteratively converted into the associated binary representation. In this process the address bit parameter is used to determine the number of bits for the binary representation. Using this binary representation the connections between the metal lines are established. In case of a `0`, the output is shorted to an inverse address bit, and vice versa.

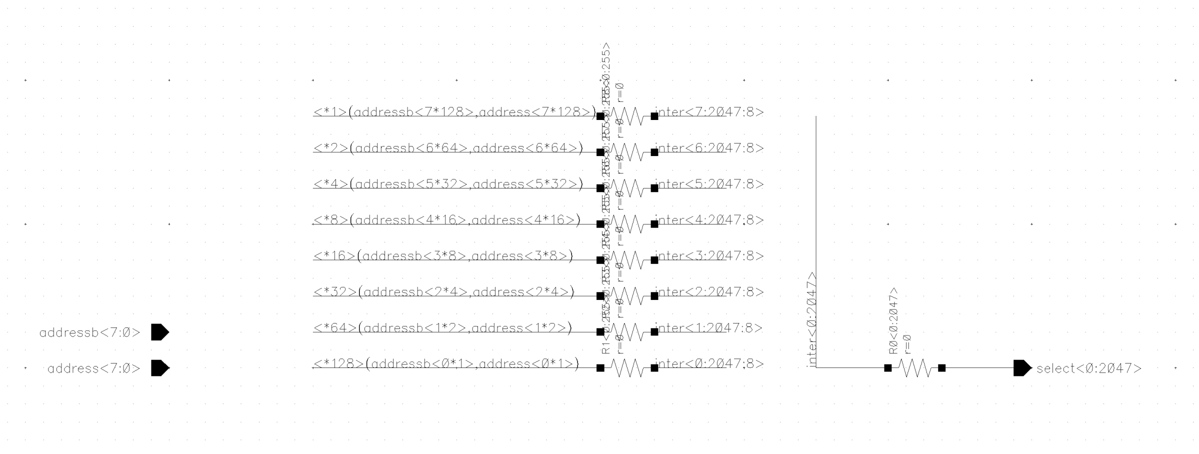
Explaining this concept an example of the number 5 decoded in a 4-bit address system is shown. When the loop calculating the output decoding reaches the 5 inside the decoding list, it will be converted into 0101. A second loop will take one character of the 0101 string for each routing line within that output. This means that the coordinates of the first routing line will be merged with the ones from the inverse address line 4, the coordinates from the second will be merged with the ones from the address line 3 and so on. Essentially this implements a simple binary representation of a given decimal decoding number by combining the according line positions into via positions.

The remaining implementation of the layout was fairly simple. One thing that was added at the end of the design process was the creation of error messages drawn directly into the design, for the cases where the address or routing positions are not specified. If either one of these position defining parameters is left empty, the decoder will not be drawn. Instead a label stating the missing parameter will be created.

Schematics

Regarding the creation of the schematics the most challenging part was implementing a working net naming creating connections between named nets using SKILL. While the order and net naming of the address inputs stay the same for all parameters, the naming of the outputs has to be adjusted based on the decoding. To achieve this, a normal forward decoding is implemented using net naming to create the desired connections. For an 8-bit decoder this is shown as an example in figure 11.

Allowing for different decodings the `select` net is renamed according to the desired order. So for example when the decoding is set to backward, the name is changed to `select<2047:0>`. Accordingly for custom decodings a large net name following the net naming conventions as well as the decoding requirements is created. To achieve these connections and finally create these nets of different names, the problem of attaching the net names to the wires had to be solved. When creating schematics by hand including nets of different names, a viable option often used to connect these is the so called patch cord "instance". Here an incoming net can be connected to an outgoing net in differing orders by stating a patch cord expression defining the connections. These patch cords are working without any problems when implemented by hand, but can not be used in PCells in a straightforward fashion. The reason for this is that patch cords are not real instances, but rather components that provide hints to the schematic editor to generate the right



addressb<7:0>
address<7:0>
<*1>(addressb<7*128>,address<7*128>)
<*2>(addressb<6*64>,address<6*64>)
<*4>(addressb<5*32>,address<5*32>)
<*8>(addressb<4*16>,address<4*16>)
<*16>(addressb<3*8>,address<3*8>)
<*32>(addressb<2*4>,address<2*4>)
<*64>(addressb<1*2>,address<1*2>)
<*128>(addressb<0*1>,address<0*1>)
r=0
inter<7:2047:8>
inter<6:2047:8>
inter<5:2047:8>
inter<4:2047:8>
inter<3:2047:8>
inter<2:2047:8>
inter<1:2047:8>
inter<0:2047:8>
inter<0:2047>
R0<0:2047>
select<0:2047>

Figure 11: Schematic view of the implemented decoder including the net names used when creating the connections.

underlying connections (Beckett, 2023). This meant that to create the connections between the nets, representing the decoding, the easiest solution was to name the nets in such a fashion that the decoding aimed for arose from the naming order. Generating these net names attached to the wires turned out to be the next challenge. To achieve the creation of these so called wire labels, which save the net name for connections, different options were explored. In the end three possible implementations were found. Interestingly all three implementations create labels that look exactly the same to the GUI user, with only one implementation actually working in PCells. These three implementations look like this.

```
net = dbCreateNet(cv "name")
wire = dbCreateLine(cv list("wire" "drawing") list(x1:y1 x2:y1))
dbAddFigToNet(wire net)

; Option 1
dbCreateTextDisplay(net wire list("wire" "label") t x1:y1 "lowerLeft" "R0" "stick" 0.0625 nil nil t
↪  nil t)

; Option 2
schCreateWireLabel(cv wire x1:y1 "name" "lowerLeft" "R0" "stick" 0.0625 nil)

; Option 3
text = dbCreateLabel(cv '("wire" "label") x1:y1 "name" "lowerLeft" "R0" "stick" 0.0625)
text~>parent = wire
```

Code 5: Different code options trying to implement the net naming in the schematic. Only option 3 is working inside PCells.

Of these implementations only the third works in combination with the PCell requirements. While the first option displays the net name as expected, all name information is lost during the check and save process of a design.

The second implementation is a direct representation of the function that is executed when using the "add name" option for wires in the schematic cellview. This function works without problems as long as the PCell is not leaving the schematic design environment. But as soon as the PCell leaves this environment, for example when executing an LVS check, an error complaining about an unknown function appears.

Using the third option all labels and connections are created and all tested simulations complete without errors. This option is also check and save compatible, resulting in no further problems when reusing this PCell in other designs.

Symbol

When it comes to the creation of a symbol for this decoder some terminal names had to be matched with the ones already calculated in the schematic. Because the code creating these pin names was already created, the implementation went rather smooth without further problems. Now that a layout, a schematic and a symbol of the decoder were created, the designs were subject to testing.

4.2.3 Verification of the address decoder PCell

Regarding the design tests, the first test executed has been a DRC of the layout for the created decoder placement. Here the PCell passes the test successfully. To verify that the PCell is DRC clean for more than some cherry picked values, the tests were repeated with a lot of different parameters. Tests at the parameter boundaries have been carried out as well, thereby demonstrating that for the normal use cases of a decoder the design appears to be stable and stays DRC clean.

After verifying the design rules for the layout, the next step was to execute an LVS check. Here the next problem surfaced. While the netlisting of the schematic, now containing the right net names, worked without any problems, the LVS aborted as soon as Calibre tried to generate a netlist of the layout representation. The created error reported that the netlisting did not find any instances in the layout, or in other words that the layout was empty. As a short look revealed that the layout in fact was not empty and that the metal lines had labels and pins attached, the LVS was restarted. Surprisingly the extraction aborted with the same error.

To finally fix this problem, resistors were added in front of the input of the decoder. The layout netlisting then was able to identify some instances and compared these with the ones in the schematic. Now the LVS completed without any errors and reported no differences when comparing the design views. Again different parameters as well as different decodings were tested to ensure that the layout and the schematic match. After this verification process of the PCell first with Calibre DRC and then with Calibre LVS the next step was to simulate the design using the schematics with the purpose of validating the connections.

4.2.4 Simulation

Creating the simulations which are used for checking the connections, a cellview with multiple voltage sources has been put together. After that these voltage sources were connected to some of the inputs. The rise and fall time of these voltages were defined in a way, that the voltage of each input would rise at a different point in time. Selecting different in and outputs allowed the simulation of different connections inside the decoder. One of these simulations is shown in figure 12.

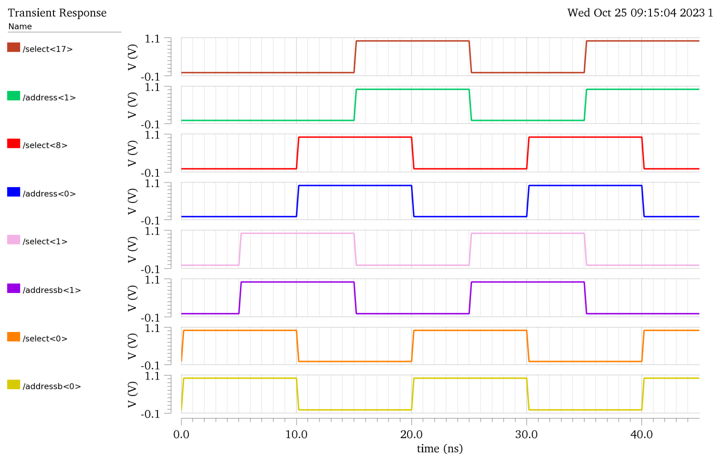


Figure 12: Simulation results for digital signals passing the decoder. Nets that are related to each other are plotted next to each other.

Here the connections as shown in figure 19 can be seen. The selection of the plotted output is based on the idea of only showing the most interesting results. Therefore the first select output, which is connected to each address input, has been plotted. In this simulation the metal length differences between outputs of lower and higher numbers in the layout, which could result in for example longer signal times, could not be shown. The reason for this is that this simulation only was executed on the schematic without considering any parasitic effects.

4.3 Flexible synapse driver array

Now that a working replacement for the via pattern in the fixed synapse driver array was created, the transformation of the driver array into a flexible PCell was executed. The goal of this transformation was to allow the user to choose a number of synapse driver outputs from a list while simultaneously allowing them to choose if they should be instantiated starting from the top or from the bottom of the array. Besides, an additional option allowing for the instantiation of a single driver row surrounded by parasitic model cells, used for faster simulations, has been implemented. The following chapters will provide insight into the creation process of this PCell. For this the explanation is divided into the different cellviews, highlighting the thought processes that went into solving the different challenges that had to be overcome during the creation process. Because the creation of this PCell did not require as many parameters as the decoder, the parameters implemented here are discussed in less detail than before. Keeping this in mind, the order of cellviews that are going to be explained will stay the same, starting with the explanation of the layout creation.

4.3.1 Layout

The creation of the scalable layout started by using the newly created decoder cell to replace the vias, previously placed in an extra step using SKILL. Here the different parameters were tested by fitting the cell to the vias and then noted for the instantiation process in the PCell. In this step the possibility of defining the routing position using mathematical expressions proved to be advantageous because the connections of the SRAM address bits are placed in a quite challenging order. For example the function defining the routing position of these addresses looks rather complex `"if(i<7 then b=0.2u*i) if(i==4 then b=2u) if(i==5 then b=1.8u) if(i==6 then b=1.6u) b"`. After figuring out the right parameters, all vias from the previous SKILL script that were used for decoding purposes could be replaced using the newly created decoder PCell. Interestingly some further vias, not used for decoding purposes but located in fixed places, were found. Because the purpose of the decoder was not to replace vias placed in a fixed pattern, the code generating these vias has been reused for the PCell creation.

Finishing the move of these vias, the next step was to move the different design parts of the driver into the PCell. To achieve this, the various designs used in the creation of the fixed driver array were inspected and instantiated inside the PCell by using the according SKILL functions. Deleting the moved designs from the old layout revealed some remaining shapes that previously had not been part of any hierarchical designs. This meant that these shapes could not be accounted for by simply moving the design parts from the old cell into the PCell. The leftovers included some metal shapes at the top and bottom of the design as well as at the sides. Here the pieces at the top and bottom were bundled up in two layout designs, one containing all the metal left at the bottom, and one containing all the metal left at the top. These leftover cells then could be instantiated from inside the PCell, while the metal lines drawn at the side were integrated directly into the PCell code, allowing for a later needed length scaling.

The last thing that had to be moved from the old layout to the new one created by the PCell were the connection labels. Here the script creating these labels for the old design was picked apart so that the labels needed were transferred into the new design. Once this transfer was completed, an LVS against the old schematic proved that everything was migrated successfully. After moving the old layout into the PCell the parameterization process began.

To achieve this, a parameter specifying the number of output synapses was added. A scaling of four drivers per step has been chosen, resulting in 8 output rows, driving 16 synapses per step. Why this decoding seemed intuitive will become clear when looking at figure 5. Here it is shown that the drivers are connected to the busses in such a fashion, that if the address of each bus increases by one, the number of instantiated drivers goes up by four. While it would be possible to only add one additional driver to one bus, this seems quite unsensible considering the symmetry of the design.

To achieve the desired scaling, the number of synapse outputs is used to calculate different parameters used throughout the design. In the decoder for example it is used to calculate the maximum number of routing outputs. Implementing this directly results in a shrinking of the decoder depending on the chosen number of outputs. A challenge faced at this point was the implementation of the option for shrinking the synapse driver array, and therefore shrinking the decoder, not only from the top down but also from the bottom up.

Because the option restricting the maximum number of outputs for the decoder always starts shrinking from the highest output, meaning the output that is only connected to the address lines, the shrinking from the bottom up, resulting in a cutting of the lowest outputs, had to be implemented using a small trick. If the decoding is switched from forward to backward, the highest and lowest output essentially switch places, `output 0` is now connected to `address`, instead of `inverse address`. If the expressions stating the positions of the address and inverse address lines are swapped, the via placement in the layout appears to be the same. The difference however is that for the decoder the highest output now is at the bottom, and not at the top, because all address lines are connected to the output at the bottom, while all inverse address lines are connected to the output at the top. Implementing this switch between forward and backward decoding, while also switching the positions of the address and inverse address lines, therefore allows the decoder to skip the lowest outputs, instead of the highest, when limiting the number of outputs. Using this trick when shrinking the array from the bottom up, an offset in the y-direction had to be applied to the cells, essentially moving the decoder to a new starting position in the layout.

This offset in the y-direction had to be implemented into nearly all inserted designs subject to change in the shrinking of the array. For the static vias the shrinking process has been implemented by changing the remnants of the old SKILL procedure, in such a way that the boundaries of the loops change with the parameter, therefore only creating the vias at the right positions. For the instances that have been implemented as simple mosaics, see Code 7, the number of created instances is altered according to the number of desired outputs. Similarly, the sizes of all previously non scaling designs, mainly the metal lines at the sides, are changed. With the scaling of the vias and cells implemented, the labels were changed too. Again the implementation was quite challenging mainly due to the naming scheme used for the `corren` and `corenres` addresses. These labels and additionally created ones, as well as their numbering scheme, can be observed in figure 13.

While this labeling may look a bit confusing at first, it has been created for the four-part design, where the addresses 0 to 255 are allocated for the upper left quadrant. Looking at it from this perspective the numbering makes much more sense. Using this information one can hopefully understand what happens to the labels if for example the first 8 drivers are

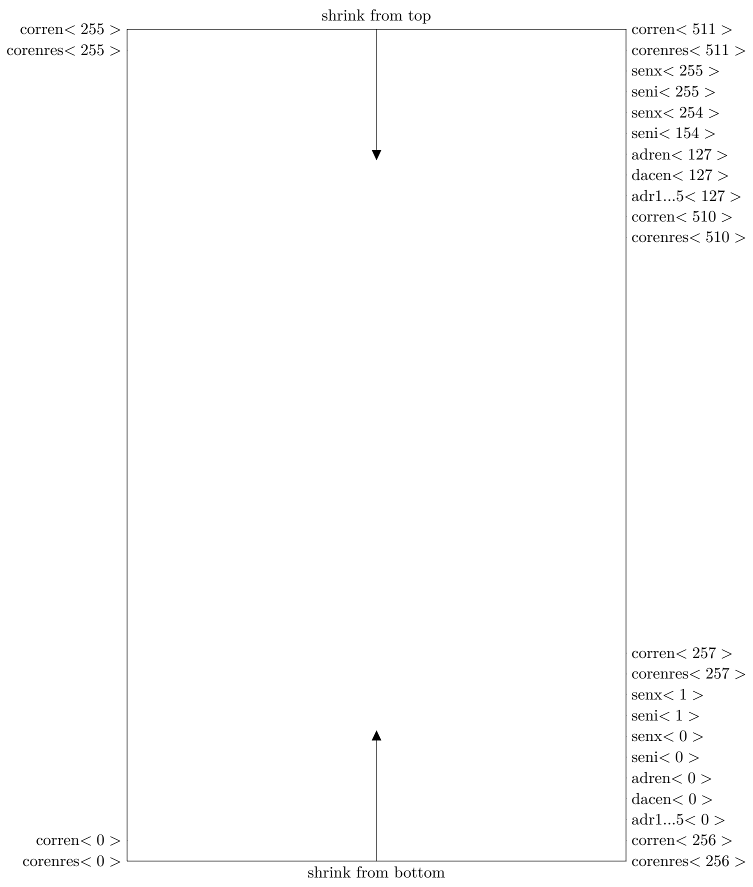


Figure 13: Names and order of the output labels used in the layout. The label position is shown in correspondence to the label number.

excluded from the design. It would be convenient when the counting in the `corren` and `corenres` address labels would start at 16 on the left and at 272 on the right. Similarly the `seni` and `senx` labels would start at 16 on the right while the `adr` labels would start at 8. The reason for this would be the possibility to investigate simulated signals of different arrays without the need to rename the simulation outputs for each array size. For implementing this behavior into the PCell the boundaries of the loops used to create these labels were changed similar to the ones placing the fixed vias. Problematically, some legacy loops used here were implemented counting down, starting with the creation of the highest label, while other loops were counting up, starting with the creation of the lowest label.

Adding the labels, the layout as described so far is capable of creating a full array as used in previous designs, as well as a shrunken array, allowing for small scale chips in the possible future.

4.3.2 Schematic

For the creation of the schematic the steps taken have been very similar to the ones in the layout. To realize the implementation of the scaling capabilities, the old schematic has been redrawn as a PCell. Here the net and wire names had to be created using the same method already explained in the decoder section with Code 5. Further, the parts of the design that stay fixed for an arbitrary number of drivers were combined into one instance summarizing the functionality of these parts. To verify the recreation and preparation of the schematic for the purpose of scalability, an LVS against the old design has been executed thereby proving once more that the connectivity implemented in the different design views stayed the same after moving to a PCell. Being able to utilize the working PCell, the parameterization process started. The parameterization process of the schematic has been easier than the one of the layout, mainly because the change in driver numbers did not result in any position or instance size changes.

To implement the shrinking functionality of the array into this PCell, the driver schematic has been altered, so that the number of instantiated driver cells changes according to the number of drivers, again in packs of 4. This has been achieved by renaming the driver instance, previously instantiated several times using the array notation, according to the "number of outputs" parameter and therefore creating a scalable number of drivers. Because of this changing number of instances the connections of the incoming and outgoing wires also have to change size. Not implementing this size change would result in mismatching connection widths between terminals and wires. For the driver inputs this is achieved by simply changing the names of the input nets according to the shrinking direction by lowering the highest or raising the lowest address number. As a result the nets inside the schematic change in size. For the outputs the naming process has been quite similar with the difference that now the output pins in the schematic had to be renamed too. Implementing these net names the option of connecting nets by name in the schematic was used, allowing a connection between the pins and terminals without the need to draw wires everywhere.

4.3.3 Symbol

For the process of creating the symbol of the synapse driver array the same arguments made during the creation of the decoder symbol can be applied. Reusing the pin labels created in the schematic resulted in already scalable outputs without the need for additional work to complete the creation of the symbol.

4.3.4 Sparce array for fast simulation

After the scaling options from the top and bottom had been implemented into the designs, the next step was the addition of the option to only instantiate a single output driver block. To achieve this, a parameter specifying which individual instance should be placed has been added.

Starting with the layout this single instance parameter defines which driver block containing 4 drivers is inserted into the layout. It is also used to calculate the position where this driver normally would sit in the full design. Using this information, the single driver block is instantiated. Keeping the goal of replacing the other drivers with different extraction models in mind, it is possible to fill the other driver slots with different designs. Here the option to specify different models, replacing the drivers, has been added to the layout. To allow this choice three string parameters specifying the library, cell and view of the desired model have been added. To simulate the instantiation of a single instance without any parasitic models a "modelcell" for replacing these drivers has been created. This so called "noinst" cell only contains the metal lines needed for net connections inside the array.

In the future this cell can also be used as a starting point for creating model cells by showcasing the net connections inside the layout. These connections have to be implemented when creating the cells containing different parasitic models. Finishing the implementation of this single instance in the layout, the option to shrink the array with only one instantiated instance has been added. Here the output labels are changed in the same fashion as before, creating all labels according to the size of the full array. While this may seem contradicting to the option of creating only labels for the single instance, it becomes a reasonable choice when keeping in mind that the model cells, to be used in the future, may be utilizing the outputs as well.

The process of adding this single driver block functionality into the schematic has been quite easy. This came as a result of the array notation, where by simply changing the instance name it was possible to instantiate only one driver block. By changing the number of drivers the nets inside the schematic had to change as well. Currently only the nets belonging to the driver block are connected. Here the possibility to connect the other nets to an arbitrary model cell has been established for future use. However, these model cells did not exist at the time of this work, resulting in currently floating outputs only used for LVS purposes.

Adding this functionality to the symbol has been achieved by simply reusing the pin labels from the schematic as before.

With these functionalities added to the driver PCell, the implementation process of the design was completed. The PCell is now able to create the full array, a shrunken array, for small scale chips in the future, and an array with only one instance for use in parasitic simulations.

4.3.5 Verification of the synapse driver PCell

To verify these PCells for the three different cellviews, some tests have been executed. The DRC tests of the layout, carried out for different array sizes and scaling directions, returned positive as expected. With all of these tests returning positive for different types of arrays the layout PCell seems to not be violating any design rules.

Further LVS tests were carried out for different array sizes. These were executed by instantiating the array as a symbol followed by the creation of the according layout view. While these tests return positive, one thing that has to be kept in mind during the execution are the labels in the layout view. The reason for this is that the labels used for net naming inside the layout are created inside the PCell. This means that they are not visible at the top layer from which the netlist is extracted. As a direct result the PCell has to be flattened to expose these labels to Calibre. While this might seem like a problem at this point it should not make any difference in the testing process of the whole chip. The reason for this is that all nets created here are routed to internal connections and therefore will be named at a higher hierarchy in the completed chip design, ultimately solving this problem.

Showcasing that DRC and LVS return positive has been the last step in the process of creating the driver array PCell. Looking at the results one can conclude that the created designs, when instantiated without shrinking, are a recreation of the old schematic and layout. It has also been shown that as long there are no identical mistakes made in schematic and layout, the shrinking of the array, as well as the instantiation of a single driver block, is possible. Trying to showcase the simulation capabilities as well as some small extractions of this PCell the next logical step has been to execute some simulations on the created designs. These simulations as well as parasitic extractions are shown in the following chapter.

5 Extraction and Simulation

Utilizing the PCells created in the course of this work, the following chapters will showcase an extraction of different possible designs by comparing the extraction times. Using some of these designs, as well as the according parasitic extractions, a comparison between a small, fully instantiated driver array and an array of the same size containing only the last four driver instances will be performed using simulations. Here it will be shown that the smaller array speeds up the extraction and simulation times, while still generating similar results.

5.1 Extraction

To perform the parasitic layout extractions of the different designs the Calibre parasitic extraction (PEX) tool was used. This tool generates a netlist containing not only all the different components, but also adding the extracted resistances and capacitances into the netlist. This netlist then is used to generate a schematic of the design containing all the extracted parasitics. Here the first speed improvements, created by extracting a smaller array, or even one with only one driver, become visible. A comparison of the different extraction times shows that the extraction time for a small array, with the size of 64 driver outputs, containing only the last four drivers is the fastest (00:08:21). The second fastest extraction time is the one seen for extracting the big array, again with only the last four drivers (00:08:50). Unsurprisingly the small array with all 64 driver outputs takes the second longest time to extract (01:03:51), while the full array takes the longest (04:20:28). The extraction times for each of these configurations as well as a small picture of the extracted layout can be seen in figure 14.

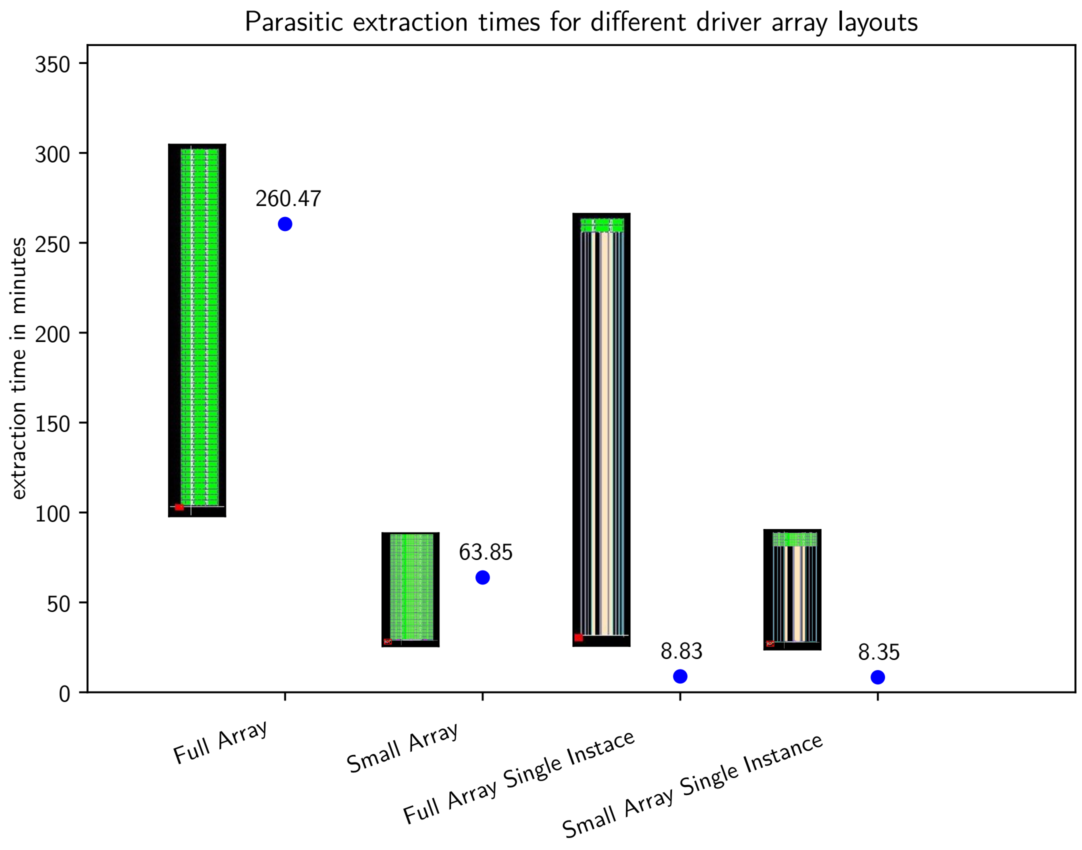


Figure 14: Parasitic extraction times for different driver array layouts. The X-Axis shows the extracted layout, while the Y-Axis shows the required extraction time. Each extraction time is plotted as a point at which the extraction time, as well as a small picture of the extracted layout can be seen.

Looking at the extraction times the case for creating faster extraction and simulation times by utilizing the flexibilities of a PCell can be strengthened. Further the goal is that it will become possible to replace the drivers in the full array with extracted parasitics to only simulate one instance, while still keeping the advantage of the sped up extraction as seen in the single instance array. With these parasitic models the goal is to further achieve a comparable accuracy.

Using the extractions from the small synapse driver array, schematics including the parasitics have been created. These schematics will be used in simulations showing an increased simulation speed for the sparse array. The four different configurations used in these simulations were the full and sparse arrays, both with and without annotated parasitics. The simulation times as well as the simulation results will be presented in the next chapter.

5.2 Simulation

In order to simulate the different synapse driver array configurations, a test bench schematic has been used. Using this schematic in combination with a simulation interface in Python, which in its core starts a Spectre simulation, allowed for a programmatic test description and complex data analysis. In this case the test bench was used with two different array configurations, first a small array, containing 32 drivers, and second a sparse small array, in which only the last four of these 32 drivers have been instantiated. For both of these instances the extracted parasitics from the previous chapter have been used too. To allow for the simulation of these drivers, their schematics, either with or without the parasitics, have been placed in the test bench one after another.

As the schematic containing the array to be simulated was placed inside the test bench, the simulation was started by importing the according netlist. The test bench code furthermore generated a series of stimuli. These included patterns to access the synapse drivers' internal SRAM in order to configure their addresses to accept and output synaptic events. The main stimulus sequence then injected randomized events into the PADI bus interface triggering the synapse drivers. After defining and pre calculating the stimuli and SRAM writes, operating parameters, like the chip voltages and on chip simulation times had to be set. Furthermore, the nets to be recorded had to be defined. If all parameters mentioned above are defined, the simulation can be executed. Regarding the simulation and especially the speed of these simulations the first differences between the arrays can be seen in figure 15.

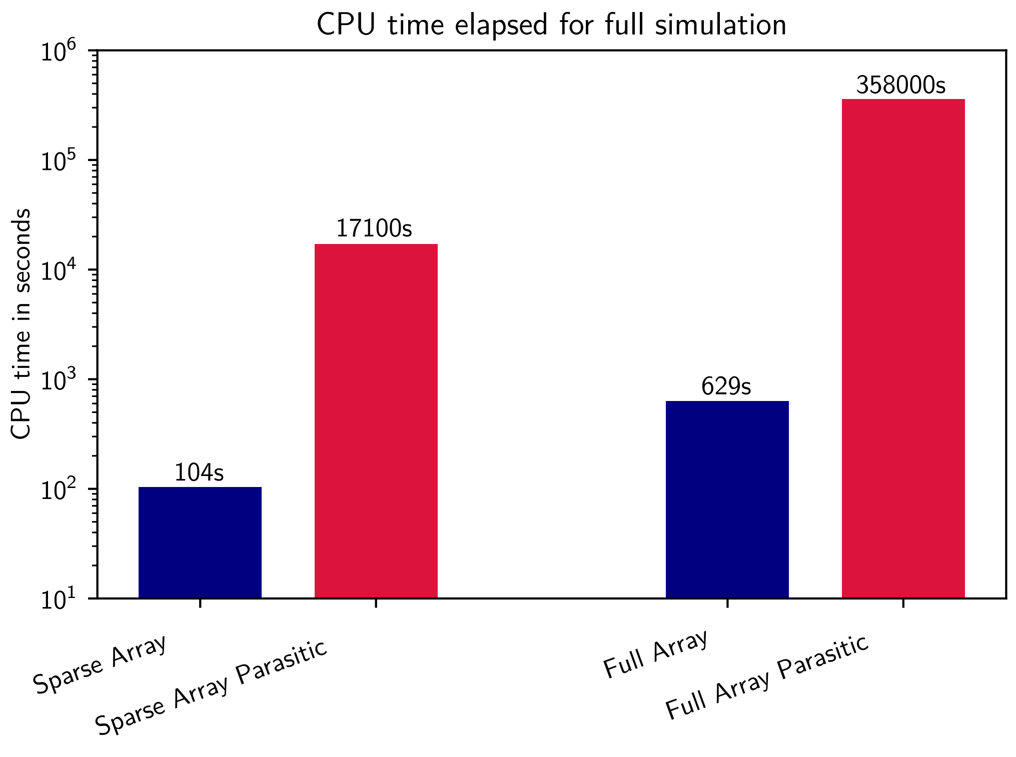
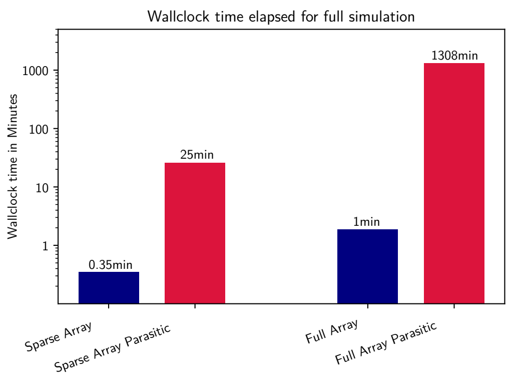
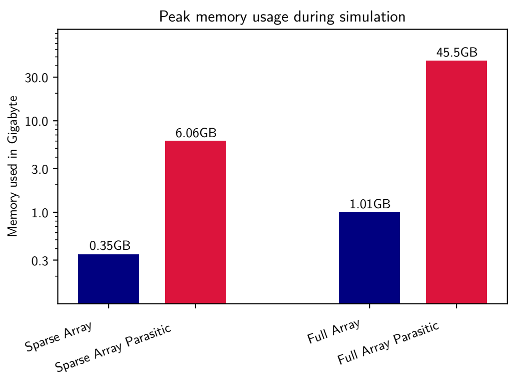


Figure 15: This plot shows a comparison of CPU time required to complete the full simulation of the different array configurations. The X-Axis is used to separate the different arrays, the Y-Axis shows the time required for the simulation in seconds logarithmically.

Looking at these simulation times it is visible, that the simulations containing the parasitic effects increase the required time drastically. This effect can be seen in both the sparse array as well as the full array. The increase in required computing time and memory usage further can be observed when looking at the real world times of the simulations as well as the peak memory usage during these simulations. The plots used to compare the different designs can be seen in figure 16.



(a) Here the wallclock time of the whole simulation in minutes is shown.



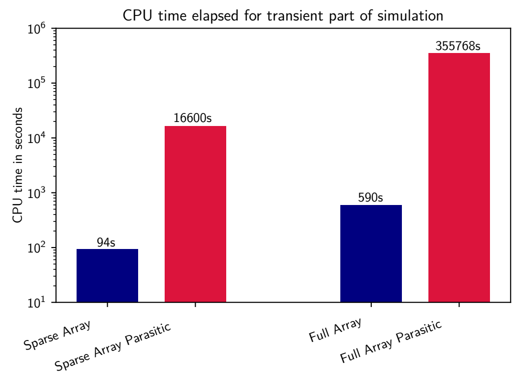
(b) Here the peak memory usage during the simulation in Gigabyte is shown.

Figure 16: These plots show a comparison of the wallclock time and peak memory usage between the different array configurations when simulating. The X-Axis is used to separate the different arrays, the Y-Axis shows the time required for the simulation in minutes or the memory used during simulation in Gigabyte logarithmically.

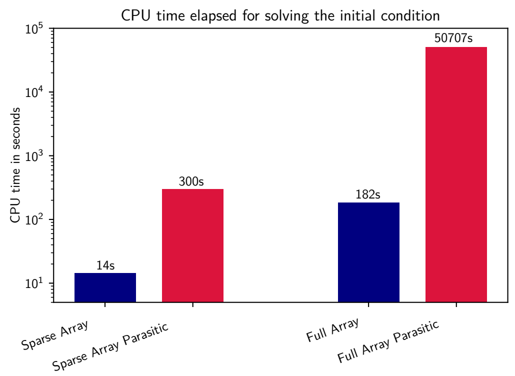
Further comparing the different parts of the simulation one can see that both the search for an operating point of the simulator, as well as the transient part of the simulation increased in computing time. Interestingly the time required for the transient simulation part increased by a factor of ~ 177 and ~ 603 between the parasitic and non parasitic simulation, while the time elapsed for solving the initial conditions only increased by a factor of ~ 21 and ~ 279 respectively. This difference can be seen in figure 17.

Comparing the CPU times required when simulating the arrays without parasitics it should be noted that the time difference between the sparse and full array is of a factor ~ 6. Interestingly the time of the transient part increases by a similar factor ~ 6.3, while the time required to solve the initial conditions increases by a factor of ~ 13. This means, that when increasing the size and therefore the complexity of a design, the search for the operation point becomes more complex faster than the actual transient simulation. This is in agreement with the array containing the parasitics, where the transient simulation time from sparse to full increased by a factor of ~ 21, while the time required to solve the initial conditions increased by a factor of ~ 169.

Taking a look at the output signals from these simulations (figure 18), one can see that full and sparse arrays, divided into the ones with and without parasitics, generate nearly identical signal forms. From these signals it can be concluded that the number of



(a) Here the CPU time of the transient simulation part is shown.



(b) Here the CPU time required for solving the initial conditions is shown.

Figure 17: These plots show a comparison of the CPU time required for the transient simulation part and the solving of initial conditions between the different array configurations when simulating. The X-Axis is used to separate the different arrays, the Y-Axis shows the time required for the simulation parts in seconds logarithmically.

instantiated drivers has next to no effect on the signals. However, there is a noticeable difference between the simulations executed with and without taking the parasitics into consideration. Looking at the DAC enable (dacen) signal of the driver output, the parasitic effects mentioned above become visible. The delayed increase, as well as the delayed decrease of the output signal can be explained by the fact that the signal generally takes more time traveling through the chip. The reasons for these effects are resistors and capacitors that can form between shapes created in the chip layout. Using these capacitors the gentler incline of the signal can also be explained. When the output signal is turned on, the parasitic model accounts for the capacitive load on for example the transistor gates. A section of this signal for all four simulated configurations can be seen in figure 18.

Looking at the address enable (adren) output of the driver a similar behaviour can be observed. The difference between the simulation of the arrays with and without the consideration of parasitic effects is clearly visible, while the difference between the two array sizes is negligible.

To further analyse the form of these signals and investigate possible input capacitances, more and different simulations are needed. These simulations are also required when possibly comparing different models for the synapse driver.

However, what can be concluded from the results shown here is that it is possible to use a PCell to generate differing implementations of a design for simulations. It further was shown that the simulation result from a synapse driver array containing only the last four drivers behaves similarly to an array of the same size containing all drivers. Here a definite speed improvement regarding the simulation and extraction times could be observed. The simulations therefore prove that the approach of simulating only parts of the design by using PCells is feasible and valid.

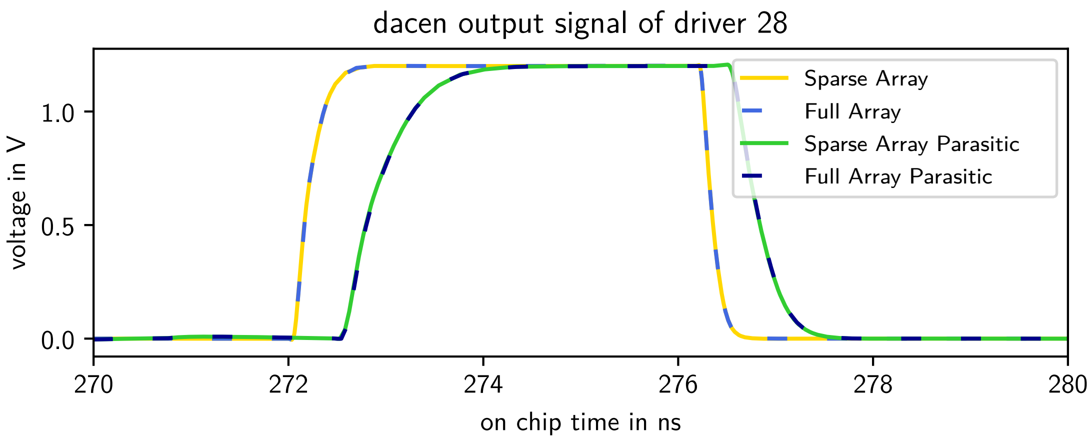


Figure 18: This plot shows the dacen output signal from the 28th driver in the simulated array. The X-Axis shows the on chip time in nanoseconds, the Y-Axis shows the voltage of the signal in volts.

6 Discussion and Outlook

This work shows the creation and simulation of a PCell of the synapse driver array used in the BrainScaleS-2 system, which is developed and maintained by the Electronic Vision(s) group. The synapse driver has been implemented as a PCell to allow for scaling of the array. Furthermore the option to create an array containing only one driver block has been implemented. As part of the creation process of this PCell another PCell, an address decoder, was created. This address decoder PCell is capable of creating a selectable decoding for a choosable number of address bits in layout schematic and symbol. One of the main features of this decoder PCell is the possibility to choose the layout positions of the metal lines by defining them as an expression. This functionality has been shown in great detail in figure 9. Utilizing the decoder PCell the synapse driver array has been parameterized. Here the options to shrink the array either from the top or from the bottom have been implemented. In addition to this scaling the option to use only one driver block has been added. Using this functionality it now is possible to not only instantiate one driver, but to also choose a model cell to replace the not instantiated driver cells. To allow for simple simulations a connection cell, consisting of only vertical metal lines has been created and used. This cell simply connects the instantiated driver cell to the corresponding signals at the top and bottom of the array.

One future goal continuing the work shown here would be to create different model cells that could replace the metal connection cell. Using these model cells, it would become possible to test different parasitic models for their extraction and simulation times, as well as compare the results for different extracted driver models. While adding this functionality into the layout now only is a question of creating these model cells, a further goal could be to allow models to be used in the schematics. Here the model cells would have to be connected to each other, simulating the different metal lengths between the drivers in the layout. This could be achieved by creating model cells representing the according layout model with predefined pin requirements. These schematic model cells then could be processed by a PCell, generating the wanted connections. Using these models the simulations shown in the second part of this work could be revisited.

Talking about the simulations shown in this work, it was demonstrated that using the created PCell it is possible to achieve a speedup in extraction and simulation times by comparing the fully instantiated small array against the small array with only one driver block. Here it was shown that the simulation results already are quite similar. However, using the already mentioned models the hope is that it will become possible to further investigate the accuracy of the simulations while keeping the increased simulation speed of the single instance array. Thus a future goal is to revisit the simulations comparing different driver models.

Taking a step back and looking at the bigger picture, this work may also be seen as a first step towards a fully scalable BrainScaleS-2 core. To achieve this goal, the parameterization as shown here has to be applied to the whole BSS-2 core. Following this route, the next step would be to create a PCell of the synapses and the neurons with parameters similar to the ones seen here in the synapse driver array. If this can be achieved, the hope is to combine these PCells into increasingly larger PCells. If that parameterization works, it would become possible to create a BSS-2 PCell that after simply asking for the number of synapses in x- and y-direction would generate the complete analog core of the chip.

7 Appendix

7.1 Decoding

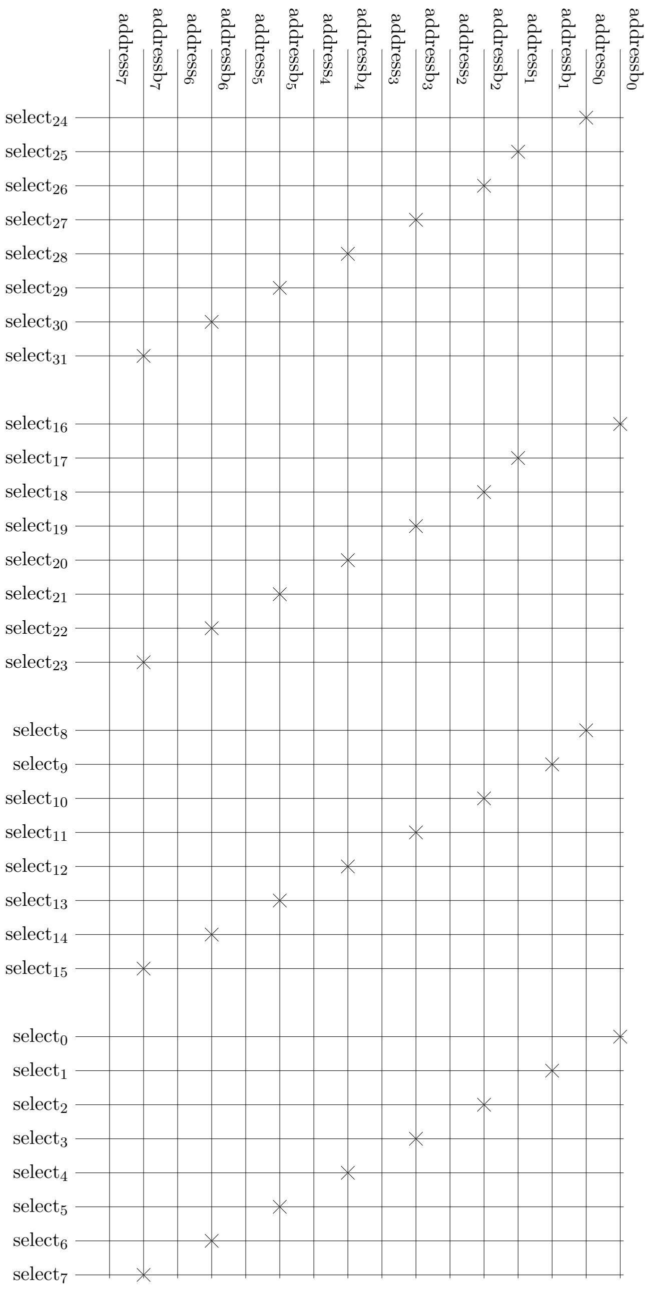


Figure 19: Decoding schematic of the decoder PCell that is created when selecting forward decoding. This is implemented in the layout and the schematic.

7.2 Code fragments

The full code is located in the Gerrit of the Electronic Vision(s) Group: `hicann-dls-fc/units/pcells`

https://gerrit.bioai.eu:9443/plugins/gitiles/hicann-dls-fc/+/3f523d5dbc49426b20541ad26bcbbd32e1996f4e/units/pcells/

```
; define the Layout
PCellLayoutId = pcDefinePCell(
    list( library cell "layout" "maskLayout")
    (
        (adBit 8)
        (customlength "boolean" nil)
        (maxBit 2**8)
        (decoding "Forward")
        (customdecoding "(0 1 2 3 4 5 6 7)")
        (metalWidth "0.18u")
        (routingLayer "M3")
        (drawAddressBus "draw")
        (addressLayer "M4")
        (addressPosition "")
        (drawInvAddressBus "draw")
        (invaddressLayer "M4")
        (invaddressPosition "")
        (drawLocalRouting "draw")
        (routingPosition "")
        (routingSpace "")
    )
    let(()
        ; here code generating the layout would be written
    )
) ; define PCell

dbSave(PCellLayoutId)
dbClose(PCellLayoutId)
```

Code 6: This code showcases the parameter definition at the beginning of each decoder cellview.

```
syndrv_top_buffered_sram = dbOpenCellViewByType("hx_synapse_driver" "syndrv_top_buffered_sram" "layout"
↪  "maskLayout" "r")
dbCreateSimpleMosaic(cv syndrv_top_buffered_sram nil 15:5+y_bottom "MY" 64-(2*sFac) 1 32 69.39)
dbCreateSimpleMosaic(cv syndrv_top_buffered_sram nil 85:5+y_bottom "R0" 64-(2*sFac) 1 32 69.39)
```

Code 7: This code is used inside the creation of the synapse driver layout. It shows the instantiation of two buffered SRAM mosaics.

```
; create a viapattern for given rectangle
procedure(createVias(xLength yLength xPos yPos layer cv)
    let((y nY i x nX k)
    y = yPos+0.5*yLength-space_to_edge_y
    nY = round((yLength-2*space_to_edge_y)/viaspace_y)
    i = 0
    while(i<nY
        x = xPos-0.5*xLength+space_to_edge_x
        nX = round((xLength-2*space_to_edge_x)/viaspace_x)
        k = 0
        while(k<nX
            dbCreateRect(cv list(layer "drawing") list(x:y x+min_via_width:y-min_via_height))
            x = x+viaspace_x
            k+=1
        )
        y = y-viaspace_y
        i+=1
    )
    )
)
```

Code 8: This code defines a function used to create a viapattern on metallcrossings with a crossection greater than 0.2 µm

7.3 Reproducibility

To allow for reproducibility of the work shown in this thesis, the states of the used software are given below.

hicann-dls-fc	3f523d5
teststand	b89eec3
container	/containers/testing/asic__c20219p37__2023-07-05__1.img

Acronyms

adren address enable.

AEL Automate Expression language.

BSS-2 BrainScaleS-2.

CMOS Complementary Metal-Oxide-Semiconductor.

dacen DAC enable.

DRC Design Rule Check.

IC integrated circuit.

LVS Layout Versus Schematic.

NMOS n-type Metal-Oxide-Semiconductor.

PCell Parameterized Cell.

PEX parasitic extraction.

PMOS p-type Metal-Oxide-Semiconductor.

SPICE Simulation Program with Integrated Circuit Emphasis.

SRAM static random-access memory.

TSMC Taiwan Semiconductor Manufacturing Company Limited.

UMC United Microelectronics Corporation.

VLSI Very-Large-Scale Integrated Circuit.

References

Barnes, T.J. (1990). “SKILL: a CAD system extension language”. In: *27th ACM/IEEE Design Automation Conference*. ISSN: 0738-100X, pp. 266–271. DOI: 10.1109/DAC.1990.114865. URL: https://ieeexplore.ieee.org/document/114865 (visited on 11/10/2023).

Beckett, Andrew (2023). *(10) Error while using LVS on a Schematic PCell - Custom IC SKILL - Cadence Technology Forums - Cadence Community*. en. URL: https://community.cadence.com/cadence_technology_forums/f/custom-ic-skill/58124/error-while-using-lvs-on-a-schematic-pcell (visited on 11/12/2023).

Billaudelle, Sebastian (2017). “Design and Implementation of a Short Term Plasticity Circuit for a 65 nm Neuromorphic Hardware System”. Masterarbeit. Universität Heidelberg.

Cadence Design Systems, Inc. (2004). *Virtuoso Parameterized Cell Reference*. 5.0. Cadence Design Systems, Inc. 555 River Oaks Parkway, San Jose, CA 95134, USA.

Cadence Design Systems, Inc. (2023a). *Circuit Simulation*. en. URL: https://www.cadence.com/en_US/home/tools/custom-ic-analog-rf-design/circuit-simulation.html (visited on 11/09/2023).

Cadence Design Systems, Inc. (2023b). *Integrated Circuit (IC) Design*. en. URL: https://www.cadence.com/en_US/home/explore/what-is-ic-design.html (visited on 11/07/2023).

Grübl, Andreas et al. (2020). “Verification and Design Methods for the BrainScaleS Neuromorphic Hardware System”. In: *Journal of Signal Processing Systems* 92.11, pp. 1277–1292. DOI: 10.1007/s11265-020-01558-7. URL: https://doi.org/10.1007/s11265-020-01558-7.

Jaeger, Richard C. and Travis N. Blalock (2011). *Microelectronic Circuit Design*. Ed. by Marty Lange. fourth. McGrawn-Hill. ISBN: 978-0-07-122199-3.

Kao, W.H. et al. (2001). “Parasitic extraction: current state of the art and future trends”. In: *Proceedings of the IEEE* 89.5, pp. 729–739. DOI: 10.1109/5.929651.

Ma, Yaoyao et al. (2023). “Extraction of Interconnect Parasitic Capacitance Matrix Based on Deep Neural Network”. en. In: *Electronics* 12.6, p. 1440. ISSN: 2079-9292. DOI: 10.3390/electronics12061440. URL: https://www.mdpi.com/2079-9292/12/6/1440 (visited on 11/09/2023).

Moore, Gordon E. (2006). “Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.” In: *IEEE Solid-State Circuits Society Newsletter* 11.3, pp. 33–35. ISSN: 1098-4232. DOI: 10.1109/N-SSC.2006.4785860. URL: https://ieeexplore.ieee.org/document/4785860 (visited on 10/31/2023).

Our World in Data (2023). *Moore’s law: The number of transistors per microprocessor*. URL: https://ourworldindata.org/grapher/transistors-per-microprocessor?time=earliest..2021 (visited on 10/31/2023).

Patni, Ashish (2021). *Virtuosity: Custom IC Design Flow – Introduction - Analog/Custom Design - Cadence Blogs - Cadence Community*. en. URL: https://community.cadence.com/cadence_blogs_8/b/cic/posts/custom-ic-design-flow-methodology-introduction-737044421 (visited on 11/07/2023).

Pehle, Christian et al. (2022). “The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity”. In: *Frontiers in Neuroscience* 16. ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2022.795876 (visited on 11/06/2023).

Pfeil, Thomas et al. (2013). “Six Networks on a Universal Neuromorphic Computing Substrate”. In: *Frontiers in Neuroscience* 7. ISSN: 1662-453X. URL: https://www.frontiersin.org/articles/10.3389/fnins.2013.00011 (visited on 11/06/2023).

Phillip E. Allen, Douglas R. Holberg (1987). *CMOS Analog Circuit Design*. Oxford University Press, Inc. 198 Madison Avenue, New York, New York 10016. ISBN: 0-19-510720-9.

Razavi, Behzad (2013). *Fundamentals of Microelectronics*. en. Google-Books-ID: zpMYAgAAQBAJ. John Wiley & Sons. ISBN: 9781118156322.

Riordan, Michael, Lillian Hoddeson, and Conyers Herring (1999). “The invention of the transistor”. In: *Reviews of Modern Physics* 71.2, S336–S345. DOI: 10.1103/RevModPhys.71.S336. URL: https://link.aps.org/doi/10.1103/RevModPhys.71.S336 (visited on 10/31/2023).

Schemmel, J. et al. (2006). “Implementing Synaptic Plasticity in a VLSI Spiking Neural Network Model”. In: *The 2006 IEEE International Joint Conference on Neural Network Proceedings*. ISSN: 2161-4407, pp. 1–6. DOI: 10.1109/IJCNN.2006.246651. URL: https://ieeexplore.ieee.org/document/1716062 (visited on 11/06/2023).

Schemmel, Johannes et al. (2020). *Accelerated Analog Neuromorphic Computing*. Tech. rep. arXiv:2003.11996 [cond-mat, q-bio] type: article. arXiv. DOI: 10.48550/arXiv.2003.11996. URL: http://arxiv.org/abs/2003.11996 (visited on 11/07/2023).

SimTech Cadence Design Systems, Inc. (2023). *EM Extraction and Finite Element Method (FEM) in Clarity 3D Solver*. en. URL: https://community.cadence.com/cadence_technology_forums/system-analysis/f/clarity-3d-solver/50917/em-extraction-and-finite-element-method-fem-in-clarity-3d-solver (visited on 11/09/2023).

Wanlass, Frank M. (1967). “Low stand-by power complementary field effect circuitry”. US3356858A. URL: https://patents.google.com/patent/US3356858A/en#patentCitations (visited on 10/31/2023).

Yu, Wenjian, Mingye Song, and Ming Yang (2021). “Advancements and Challenges on Parasitic Extraction for Advanced Process Technologies”. In: *Proceedings of the 26th Asia and South Pacific Design Automation Conference*. ASPDAC ’21. New York, NY, USA: Association for Computing Machinery, pp. 841–846. ISBN: 9781450379991. DOI: 10.1145/3394885.3431626. URL: https://doi.org/10.1145/3394885.3431626 (visited on 11/09/2023).

Acknowledgements

To start of my acknowledgements I would like to thank PD Dr. Johannes Schemmel for providing the opportunity to write my Bachelor's thesis in the Electronic Vision(s) Group under his supervision. Regarding the supervision, I would also like to thank Prof. Dr. Peter Fischer for conducting the second review of this thesis.

Special thanks go to Sebastian Billaudelle, who supported me during my work and introduced me to the world of chip design. Similarly I would like to thank Philipp Dauer and Yannik Stradmann, for helping with all different kinds of questions.

Further I would like to thank Laura Arnold and Philip Quicker, for any support since the first semester, and especially for proofreading this thesis. On the topic of proofreading I also have to thank Penelope Hoffmann, and Heike Schlatterer.

I would like to thank Jakob Kaiser, Philipp Spilger, Josha Ilmberger and Eric Müller, for the amazing learning opportunities regarding the backend hardware of the BSS systems, as well as the whole Electronic Vision(s) Group for an amazing work environment.

Finally, I would like to thank my family, especially my parents Brigitte Arth-Haas and Georg Haas, who supported and encouraged me through the course of my studies.

The work carried out in this Bachelor Thesis used systems, which received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under the Specific Grant Agreements Nos. 720270, 785907 and 945539 (Human Brain Project, HBP).

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

[illegible]

Heidelberg, den 29.11.23, Kaspar Frieder Haas