Internship at the electronic vision(s) Group

Kaspar Haas

31.05.23-06.09.23

SKILLfully creating PCells

In the world of chip and circuit design, it is often desirable to create flexible circuits that change according to a given parameterization. To achieve this, the schematics as well as the physical layouts of these circuits need to be generated in a programmable fashion. For this purpose, Cadence®Virtuoso®allows the automated generation of design data via a proprietary scripting language, SKILL, and the "PCell" framework. The following manuscript shall become a starting point and somewhat of a documentation of the process creating such PCells. Therefore it will introduce all concepts and procedures required to implement such PCells for schematics, layouts and symbols. This will be demonstrated on the example of a scalable current mirror. It will also demonstrate that the generated PCells work as intended by completing the LVS and DRC checks as well as simulating the mirror.

Contents

1 Introduction

During the course of my work for the electronic vision(s) group, a SKILL script for creating the schematic, layout, and symbol of a current mirror was created. A PCell in Cadence®Virtuoso®, is a parameterized cell of an instance, that when inserted into other designs can be parameterized in an arbitrary way. Hereby the parameters can vary from, for example, the simple number of instances created, up to their sizes in layout, or simple things, like if terminal names of the instance should be visible. To get used to the concept of PCells and to show how a PCell is created we designed a current mirror using a SKILL script. SKILL is the proprietary coding language used by the Cadence®programs. It allows for communications with the design software as well as with the design database. When combined with the according software it is a powerful tool for designing and handling PCells. To help create PCells in upcoming designs the following pages will give an insight in how to design PCells using SKILL scripts while being a showcase of some use cases by using the example of the current mirror created.

This report will follow the structure of creating a PCell step by step and also view by view. At each step there will be an example how the shown functions can be implemented. Additionally, further resources like function documentations published by Cadence®will be given accordingly.

2 Background

2.1 SKILL Language

To create a PCell we used the Cadences®coding language SKILL. SKILL is a programming language using a mix of lisp and C-like syntax. Therefore it is possible to write in lisp syntax using for example: `(plus 1 2)` which would result in 3, as well as in C-like syntax `plus(1 2)`, which would also result in 3. To keep the code readable and consistent we decided to stick to the C-like syntax. To make the code executable by Virtuoso®, we created a SKILL file containing all the information needed for Virtuoso®to create the PCell. This file follows the following structure.

`(plus 1 2)`

`plus(1 2)`

2.2 SKILL File

To create a PCell using the SKILL file one starts by telling Cadence®in which library and under which cell name the PCell is to be created. In our code these values are set to the `examplelib` and `example_cell`.

```
; library/cell of PCell
library = ddGetObj("examplelib")
cell = "example_cell"
```

To actually create the PCell the function `pcDefinePCell()` from Cadence®is used. In

`pcDefinePCell()`

this function the first argument expects the library and cell as stated above followed by the name and the viewtype of the cell to be created. The second argument is a list containing all the Parameters with their respected defaults. The function returns a database object that will be referenced later, hence why it is stored in a variable here. After the parameter definition, the code generating the structures inside the PCell is stated. For more information also see Cadence Design Systems (2004b, pp. 4–11), Cadence Design Systems (2002, p. 309) or Cadence Design Systems (2004c, p. 195). After this function we use `dbSave()` and `dbClose()` to save and close the created cell. This procedure is repeated for all cellviews that should be created when running the SKILL file. An example could look like this.

`dbSave()`
`dbClose()`

```
; layout definition
PCellLayoutId = pcDefinePCell(
    list( library cell "layout" "maskLayout")

    ; parameters of PCell
    (
        ;initializing parameters
    )

  ; code generating the PCell

)

dbSave(PCellLayoutId)
dbClose(PCellLayoutId)
```

Now that all the code that generates different view types has been stated, the code creating all the parameters and their behaviors for the defined cell and library follows. After the creation of the parameters some simulator information is passed to the program. If you are interested in these simulator options you can find more information at Cadence Design Systems (2005, p. 87). The respective code example looks like this.

```
;  definition of CDF parameters
let(
        ; searching the cell name
    ( cellId cdfId )
    unless( cellId = ddGetObj( library~>name cell )
                ; error, if cell not in library
        error( "Could not find cell %s." cell )
    )
    when( cdfId = cdfGetBaseCellCDF( cellId )
        cdfDeleteCDF( cdfId )
    )
        ; create standard cell
    cdfId = cdfCreateBaseCellCDF( cellId )

    ; code setting parameters

    ;;; Simulator Information
    cdfId->simInfo = list( nil )
    cdfId->simInfo->UltraSim = '( nil )
    cdfId->simInfo->ams = '( nil )
    cdfId->simInfo->auCdl = '( nil )
    cdfId->simInfo->auLvs = '( nil )
    cdfId->simInfo->cdsSpice = '( nil )
    cdfId->simInfo->hspiceD = '( nil )
    cdfId->simInfo->hspiceS = '( nil )
    cdfId->simInfo->spectre = '( nil )
    cdfId->simInfo->spectreS = '( nil )

    cdfSaveCDF(cdfId)
)
```

2.2.1 CDF parameters

cdfCreateParam()

To create the parameters of the PCell the function cdfCreateParam() is used. This function expects a number of different arguments which define the properties of the created parameter. Hereby the first parameter specifies the cdfDataId, which when using the code from before is given by cdfID, for which the parameter should be created. The function then requires two additional arguments, the name and the type. All other arguments are optional and help to further define the function. Some of the arguments used by us are defValue, which specifies the default value for the given parameter. The parameter units specifies the unit of the inserted parameter. This is especially helpful when working with lengths or parameterized sizes. Here one can specify the data type as string so a value with a respective unit can be inserted and then the option parseAsNumber can be activated which results in a converted number according to the specified unit. If the parameter is supposed to be kept as a string this can be achieved by defining the default as a string when retrieving it in the specified cellview. Otherwise, the parameter has to be stated as a number with units. What is meant by this will become clear when looking at the initializing process of parameters at the beginning of the cellview definition. Also quite important is the optional parameter prompt which specifies the text or name of the parameters displayed in the user interface when inserting the cell into a design. Another useful option is the possibility to define callbacks. In this argument functions that will be executed every time the parameter value is altered can be stated. In our case, we use this to read out the value inserted by the user and apply a constraint according to the design constraints of a cell. All of these arguments are stated using ?argumentname argumentvalue. For a complete list of the possible parameters it is advised to look into Cadence Design Systems (2005, p. 142). In our code three types of parameters were used. An integer, a string parsed as a number and a boolean. An example of each type of parameter definition is given below.

```
    ; Create settable parameters
    ; transistor multiplier
    cdfCreateParam( cdfId
        ?name "Multiplier"
        ?type "int"
        ?prompt "Mult"
        ?defValue 1
        ?callback "if( cdfgData~>Multiplier~>value < 1 then
            cdfgData~>Multiplier~>value = 1)"
    )

    ; input width
    cdfCreateParam( cdfId
        ?name "WidthIn"
        ?type "string"
        ?prompt "Input Transistor Width"
        ?defValue "200n"
        ?parseAsCEL "yes"
        ?parseAsNumber "yes"
        ?units "lengthMetric"
    )

    ; visable Pins names
    cdfCreateParam( cdfId
        ?name "VisPin"
        ?type "boolean"
        ?prompt "Visable Pins"
        ?defValue t
    )
```

For further examples please refer to Cadence Design Systems (2005, pp. 185–189).

2.2.2 Initializing parameters

As mentioned above when using the function `pcDefinePCell()` 2.2 a list containing the parameters with their respective default values has to be stated. Hereby only the parameters actually used have to be initialized, the parameters not used are optional. To keep the code consistent one can state all parameters at the beginning of every cellview. If on the other hand one wants to be able to specify different parameters for each cellview one can only state the respective parameters for the cells. We would recommend the first option. An example could look like this.

```
        ; layout                        ; schematic/symbol
    (                               (
        ( Multiplier 1 )                ( Multiplier 1 )
        ( WidthIn 200n )                ( WidthIn "200n" )
        ( VisPin t )                    ( VisPin t )
    )                              )
```

Here the mentioned difference between the initialization of the transistor width regarding the string or number properties can be seen. In the schematic view the value is needed as a string, therefore the default value is stated as a string. In the layout view we want the numerical value, remembering `parseAsNumber` with respect to the units from above, which is why the default is stated as a number and not as a string. For further information about the parameter list search the same sources as mentioned for the pcDefinePCell() 2.2 function, for example Cadence Design Systems (2004c, p. 197).

2.2.3 Layout

When creating the layout view one uses the SKILL script to create shapes as if they were drawn by hand. This means that one doesn't have to think much about connectivity and all connections between the layers work as expected. To create such rectangles or shapes on the layout the function dbCreateRect() is used. This function expects three arguments: The cellview in which the rectangle is to be created, a list containing the layer purpose pair, and a list containing the coordinates of two diagonal points of the rectangle. The function definition can be found in the documentation provided by Cadence Design Systems (2004a, p. 379). Pins in our example are also created by using dbCreateRect() with the layer purpose list("M1" "pin"), where the labels are created by dbCreateLabel(). This function expects eight arguments: The cellview, the layer purpose pair, the position, the name, the orientation, the justification, the font and the height. The documentation can be found at Cadence Design Systems (2004a, p. 384). Two examples of these functions follow.

dbCreateRect()

dbCreateLabel()

```
dbCreateRect(pcCellView list("M1" "drawing") list(0.15:0.435 0.32:0.72+LI))

dbCreateLabel(pcCellView list("M1" "pin") x+0.05:y+0.05 name "centerCenter" "R0" "stick" 0.05)
```

In cases of repeating layouts, it can happen that the same part has to be drawn over and over again. Because of that, it can be helpful to define a function drawing the repeated part of the layout. To define a callable function the function procedure() is used. In this the name is stated followed by the used parameters in brackets. After the parameters the function body is stated. The defined function works as expected and is called like every other predefined function. An example of this function definition would be the function we use to create the pins in the layout.

procedure()

```
procedure( DrawPinLayout(x y name)

    dbCreateRect(pcCellView list("M1" "pin") list(x:y x+0.1:y+0.1))
    dbCreateLabel(pcCellView list("M1" "pin") x+0.05:y+0.05 name "centerCenter" "R0" "stick" 0.05)

)
```

Further examples of using these functions can be found in our implementation of the current mirror. Here one has to pay attention to the characteristics of how SKILL handles function definitions. In theory, it is possible to define the function in the document that generates the PCell, but when doing so, either the generation of the PCell throws an error, when the function is defined in the cellviewcode, or the LVS and DRC throws an error, when the function is defined globally. To overcome these errors, one has to create a helper file containing the functions, which is loaded with the library using the libInit file. This file is stored in the library and executed the first time the library is accessed. Both of these files can be found in the appendix. For further documentation please refer to Cadence Design Systems (2004c, p. 203).

2.2.4 Schematic

To create the schematic part of the PCell, the function `pcDefinePCell()` 2.2 as explained in chapter 2.2 is used again with the according parameters. When creating the schematics of a circuit with SKILL the connections between the different parts become a lot more complex than simply drawing rectangles at the right layers. Now one has to instantiate the different components of the circuit and connect them to each other using wires. Here it is indispensable to make sure that different terminals of the components are connected to the right nets. To achieve this and to create and connect pins and terminals to according nets, the according net must first be created by the SKILL script. Here the question "How to insert an instance into a cellview?" comes to mind. To instantiate an instance it first has to be loaded as a database object. To achieve this we use the function `dbOpenCellViewByType()` which expects three arguments. The first one being the name of the library in which our instance was saved, the second one being the name of the instance and the third one being the view that shall be opened. If one saves the return of this function, the database entry, also see Cadence Design Systems (2004a, pp. 348–350), it can be used as a parameter for the function `dbCreateParamInst()`. This function creates an instance specified in its second argument inside the cellview specified in the first argument. When creating the instance the function expects three more parameters. The third parameter of the function is the name of the instance given by the program. This entry can be set to `nil`, which assigns a unique name to the instance. The fourth argument specifies the position of the created instance with `x:y` while the fifth argument specifies the orientation. This function is capable of more arguments than listed here, but only the ones stated are going to be used. Particularly it is possible to define parameters of the instance to be created. This for example enables the initialization of PCells in PCells. If you are interested in the other parameters you can find their documentation in Cadence Design Systems (2004a, pp. 400–401). A small code example for the creation of an instance would look like this.

`dbOpenCellViewByType()`

`dbCreateParamInst()`

```
; load the transistor for later instantiation
NMOSMaster = dbOpenCellViewByType("gpdk" "nch_mac" "symbol")

; create start transistor
Inst = dbCreateParamInst(pcCellView NMOSMaster nil 0:0 "MY")
```

As explained above we now have to create nets connecting the different instances and pins of the circuit. To achieve this the function `dbCreateNet()` is used. This function expects two arguments, the cellview in which the net is to be created and the name of the net. It returns the created net. For more information Cadence Design Systems (2004a, p. 472) can be consulted. To create a connection between two instances wires are

`dbCreateNet()`

added to the created net. Therefore the function dbAddFigToNet(), which expects two arguments, the figure added to a net and the respective net the figure is added to, as well as the function dbCreateLine() are used. Here the function dbCreateLine() 2.2.4 expects three arguments. The cellview in which the line is added, a layer purpose pair and a list of two coordinates specify from where to where the line is to be drawn. For a deeper documentation look into Cadence Design Systems (2004a, pp. 491+376). Combining these functions, creating a net and adding a line to it looks like this.

dbAddFigToNet()

dbCreateLine()

```
; create the gnd net
gnd = dbCreateNet(pcCellView "gnd")
gnd~>sigType = "ground"         ; change signaltype off the net

; add wires to net
dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.25:0.1875 -0.25:0.5)) gnd)
```

Here we also added the statement gnd~>sigType = "ground". This has to be done for all nets which carry a signal of a different type than signal. This code edits the signaltype of a net and therefore also the signaltype of connected terminals and pins.

To conclude the connection between instances one has to create a connection between the terminals of an instance and the net the wires are attached to. If a wire of a created net already exists, one can simply use the function dbCreateConn(). Here one specifies three arguments: The net the terminal should be connected to, the instance the terminal belongs to and lastly the terminal itself. To find the terminal of an instance we use the function dbFindTermByName(). Here the two arguments passed to the function are the master cell of the instance as well as the terminalname. To bypass this terminal search there is an easier way to connect to the terminals. SKILL possesses a function called dbCreateConnByName(). In this function one simply states the net, the instance and the terminalname. Both options generate the same result. The documentation of these functions can be found in Cadence Design Systems (2004a, pp. 487+514). A short example of connecting a terminal to a stated net would look like this.

dbCreateConn()

dbFindTermByName()

dbCreateConnByName()

```
dbCreateConn(gnd InputFETInst dbFindTermByName(NMOSMaster "S"))
dbCreateConnByName(gnd InputFETInst "S")
```

Now that we have connected the instances to the nets and from one to each other the last thing that needs to be added are the pins to the outside. For this it is advised to load a pin as an instance before using dbOpenCellViewByType() 2.2.4. Now we create a terminal for the pin using dbCreateTerm(). In this function we state three arguments. First the net for the terminal, then the name of the terminal and lastly the direction of the terminal to be created. The documentation can be found at Cadence Design Systems (2004a, p. 481). After creating the terminal and attaching it to the specified net, we now attach a pin to the terminal using dbCreatePin(). Here we have to specify three arguments. The first one being the name of the net the pin shall be connected to, the second one being an

dbCreateTerm()

dbCreatePin()

instance used as pin, here we use dbCreateInst() with the loaded master of a pin from above, and lastly the name of the pin. For more information see Cadence Design Systems (2004a, p. 484). In our example the code for loading and creating a pin looks like this.

dbCreateInst()

```
; load input pin for use
pinMaster_i = dbOpenCellViewByType("basic" "ipin" "symbol")     ; load pin
; connect ground pin to net
term = dbCreateTerm(gnd "gnd" "input")                        ; create terminal to connect to
gnd_pin = dbCreatePin(gnd dbCreateInst( pcCellView pinMaster_i "" -0.5:0.5 "R0") "gnd_pin")
```

How these functions were connected to create our example of the current mirror can be seen in the full code in the appendix.

2.2.5 Symbol

Creating the symbol of an instance follows the same basic structure as both viewtypes before. Again, we start by using the function pcDefinePCell() 2.2 followed by stating the parameters. In principle, one can just draw a rectangular box and add some terminals to create a working symbol of an instance. But by doing so, it can become quite confusing when adding more and more instances into a circuit. Because of that, the following chapter helps to understand what options one can exhaust when drawing a symbol. It is also possible to open another symbol and add it to the cell or to modify it to your liking. To do so dbOpenCellViewByType() 2.2.4 in connection with dbCreateParamInst() 2.2.4 is used in the same fashion as in the schematic 2.2.4. A short example would look like this.

```
; load the symbols for later instantiation
InputFET = dbOpenCellViewByType( "examplelib" "current_mirror" "symbol_in")

; create input transistor
InputFETInst = dbCreateParamInst(pcCellView InputFET nil 0:0 "R0")
```

Then, one also has to define all the nets and terminals, again quite similar to the work done in the schematic 2.2.4. A net is created using dbCreateNet() 2.2.4. According to the signaltype the type is changed using `gnd~>sigType = "ground"` and the terminal and the pin are connected using dbCreateTerm() 2.2.4 and dbCreatePin() 2.2.4. One difference is that in the symbolview the symbol for pins conventionally is another one, which is why the `sympin` is loaded for input and output. Here it can be desirable to set some pin parameters. This can be achieved by using for example `PinMaster~>shapes~>isOverbar = nil` or `PinMaster~>shapes~>justify = "centerLeft"`. All together the nets, terminals and pins are created like this.

```
    ; create the gnd net
    gnd = dbCreateNet(pcCellView "gnd")
    gnd~>sigType = "ground" ; change signaltype of the net
    ; create terminals
    ; load master pin
    PinMaster = dbOpenCellViewByType("basic" "sympin" "symbol")

    ; setting pin parameters (overbar, justification)
    PinMaster~>shapes~>isOverbar = nil
    PinMaster~>shapes~>justify = "centerLeft"

    ; connect terminals to net
    ; gnd terminal
    term = dbCreateTerm(gnd "gnd" "input")
    gnd_pin = dbCreatePin(gnd dbCreateInst( pcCellView PinMaster "" 0:-0.375 "R0") "gnd_pin")
```

Other functions useable in the symbolview are for example `dbCreateLine()` 2.2.4 with which it is possible to draw lines in the symbol. Here the layer purpose pair for creating the typical green symbol lines is `list(231 "drawing")`. Similarly, it is also possible to use `dbCreateRect()` 2.2.3 with the same layer purpose pair, or, as another example, with `list(236 "drawing")` to create a selection box for inserting the symbol. One can use these functions in the same way as shown in the schematic 2.2.4 or in the layout 2.2.3.

```
; draw line
dbCreateLine( pcCellView list(231 "drawing") list( 0.5:-0.25 0:-0.25))

; create selection box
dbCreateRect(pcCellView list(236 "drawing") list(-0.25:-0.375 0.25+0.5:0.375))
```

Another nice thing about creating nets and terminals using the PCell is that one can build buses for in- and outputs and that they are able to work as expected. To create a bus connection one simply has to name the net created with `dbCreateNet()` 2.2.4 according to the bus naming conventions. An example of this would be an outputnet, that grows from one line to j lines. To implement this one could use.

```
out = dbCreateNet(pcCellView strcat("out<0:" sprintf(nil "%d" j-1) ">"))
```

Here the `strcat()` is used to bind strings together to one string, and the function `sprintf()` is used to take the integer saved in j and create a string of j-1. For documentation see Cadence Design Systems (2022).

`strcat()`

`sprintf()`

With these functions you now should have everything you need to write a PCell creating Layout 2.2.3, Schematic 2.2.4 and Symbol 2.2.5 for an instance of choice.

To give an example of how these functions and definitions can be used we will show the thoughts that went into creating a PCell for a current mirror, as well as how these ideas were implemented. To begin, a short recap of what is meant when talking about a current mirror will be given.

3 Current mirror

A current mirror as seen in Figure 1, represents a current controlled current source in which the current flowing through I_{in} is mirrored in I_{out_i}.

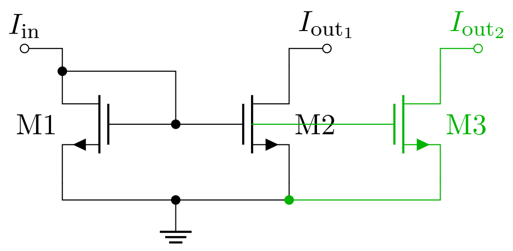

Figure 1: current mirror consisting of two NMOS transistors with a scalable transistor in green

If all of the transistor values like length, width and the multiplier are the same, the current flowing through I_{out} is exactly the same as the current flowing through I_{in}. If now for example the multiplier of the output is set to 2, the current through I_{out} is doubled. Depending on the geometry of the input and output transistors, the flowing current can be calculated using equation 1.

$$I_{\text{out}} = \frac{\left(\frac{W}{L}\right)_{\text{out}}}{\left(\frac{W}{L}\right)_{\text{in}}} \cdot I_{\text{in}} \tag{1}$$

The nice thing about the current mirror is, that one may instantiate multiple output stages. This is illustrated by the green transistor M3 in Figure 1. This “cell” of an output transistor can be added like shown over and over again. It meets the same expectations according the I_{out} as was explained above. Because of this modularity, the current mirror was chosen as an example for a parameterized cell. The goal was to be able to insert the current mirror in layout, schematic and as a symbol while making use of the parameters: number of outputs, channel geometry of in- and output and multiplier, also known as fingers, of in and output. To achieve this a PCell as explained above was used.

3.1 Implementation

In the process of implementing, we followed the order of actions as explained above. The whole code can be found in the appendix 6.

3.1.1 Parameters

The Parameters that we used to parameterize the current mirror were the number of output mirrors saved in `NumberMirrors`, the multiplication between input and output, also known as fingers of the transistor `MultIn` and `MultOut`. Further the geometry of the input and output transistor `WidthIn, LengthIn, WidthOut, LengthOut`, a boolean, letting us choose between visible and invisible terminal names in the symbol view `TermVisisble` and lastly, if the outputs should be drawn as a bus, output Bus. Of these variables the first three were chosen as integers, because it is for example not possible to draw half an output. For these parameters also a callback was written, only allowing for values greater than one. Because for the schematic view a string is needed to be passed on to the instances, the variables for the transistor geometry were defined as strings. We also added the ?parseAsNumber "yes" option to convert the strings into floats used in the layout. The last two variables are simple booleans with nothing special attached to them.

3.1.2 Layout

In the layout view the parameters were read in and processed as needed. Since we find a reappearing instance, the transistor, a function DrawTransistorLayout(x y W L m) was defined. This function draws a transistor starting with the source connection at x y, according to the specified parameters, width W, length L and the option if the transistor is the multiplier part or not, m. According to these values all sizes are calculated as can be seen in the appendix 6. Using this function the remaining work in the layout consists mostly of drawing the input and output pins as well as connecting them via metal lines. For creating the pins we also defined a function DrawPinLayout(x y name), which draws a pin starting with the left corner at x y and giving it a label with the name. Another tricky part was creating the loops calling the DrawTransistorLayout(x y W L m) 3.1.2 function with the right position and size values. In the end, a boundingbox according to the transistor sizes, multipliers and number of outputs is created around the whole instance.

DrawTransistorLayout(x y W L m)

DrawPinLayout(x y name)

3.1.3 Schematic

After loading the parameters we start by adding the input transistor as an instance.

Because we use the same master transistor for the in and output, we defined a function called InsertFET(X Y Orientation W L M), which opens the transistor to be added. It adds a transistor at X Y with a stated Orientation. It also directly sets the multiplier M as well as the channel geometry W L. To achieve this the database address of the added transistor is opened with cdfGetInstCDF(). This function accepts one argument and returns the effective CDF description of an instance. For further information refer to Cadence Design Systems (2005, 144 or 114). After saving the return value, we set the simM, so the simulated multiplier as well as the totalM, so the total multiplier to M. Because our multiplier is stored as an int, see 3.1.1, it has to be changed to a string using sprintf() 2.2.5. After setting the multiplier of the transistor we also set our channel geometry by setting w and l to the imported parameters. The function ends with a call of the created instance, to return the database object of the instance.

InsertFET(X Y Orientation W L M)

cdfGetInstCDF()

After adding our input transistor, the connecting nets as well as the wires connecting the instances and their terminals were added as explained in 2.2.4. Here one has to keep in mind that the signaltype of the ground net has to be changed. If our input would have been connected to vdd, the signal also would have been changed. After adding the connection to the input transistor a loop creating the output transistors, according to the parameter `NumberMirrors` 3.1.1, is started. Here the transistors are added using our function `InsertFET(X Y Orientation W L M)` 3.1.3. The loop also contains all the code creating the connections to the transistor as explained in schematic2.2.4. One peculiarity here is that the first mirror has to connect the gates to the input, which is why an `if()` statement is used. The code used in the loop also directly creates the output pins using, among other functions, `dbCreatePin()` 2.2.4.

The input pins for the input and ground nets are created right after the loop following the explanation in 2.2.4.

3.1.4 Symbol

The creation of the symbol is again started by importing the relevant parameters. Because we wanted the symbol to be easily recognizable we saved a picture of an input and output transistor in the files "symbol in" and "symbol out". These files contain exactly what the names suggest. To load these views we again use the function `dbOpenCellViewByType()` 2.2.4 and add them using `dbCreateParamInst()` 2.2.4 as described above. After that, the ground and the input nets were created as explained in 2.2.4. Because we wanted the possibility to make the writing over the terminals disappear, the parameter `TermVisisble` was added. Depending on this parameter the `sympin` is loaded with or without the written labels. The selected pins were then added to nets using `dbCreatePin()` 2.2.4. The code snippets `PinMaster~>shapes~>isOverbar = nil` and `PinMaster~>shapes~>justify = "centerLeft"` are used to set the option of overlined labels, as well as the anchor point of the label.

The second option we wanted to add to the symbol was the option to create an outputbus of all outputs. For this the parameter `Bus` is used. If this parameter is turned off, the scaling of the outputs is handled similarly to the scaling in the schematic. If the option is turned on only one output transistor is drawn and the net attached to output is named using `strcat("i_out<0:" sprintf(nil "\%d" NumberMirrors-1) ">")`. This creates an outputbus that in the case of 4 outputs would be named like this `i_out<0:3>`. At the end, a green box around the instance as well as a selection box are created.

3.2 Created Views

The following pictures show the cells that were created using our SKILL script.

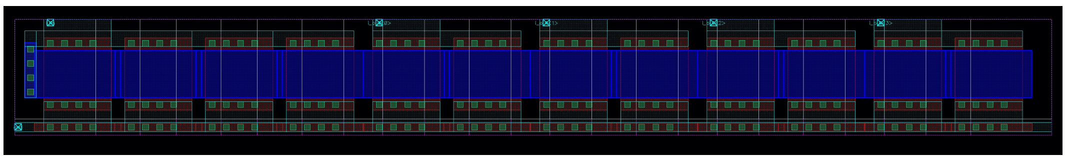

Figure 2: Result of the PCell in Layout for Number Mirrors = 4, Input Multiplier = 4, Output Multiplier = 2, W_{in} = 1µm, L_{in} = 700nm, W_{out} = 1µm, L_{out} = 700nm

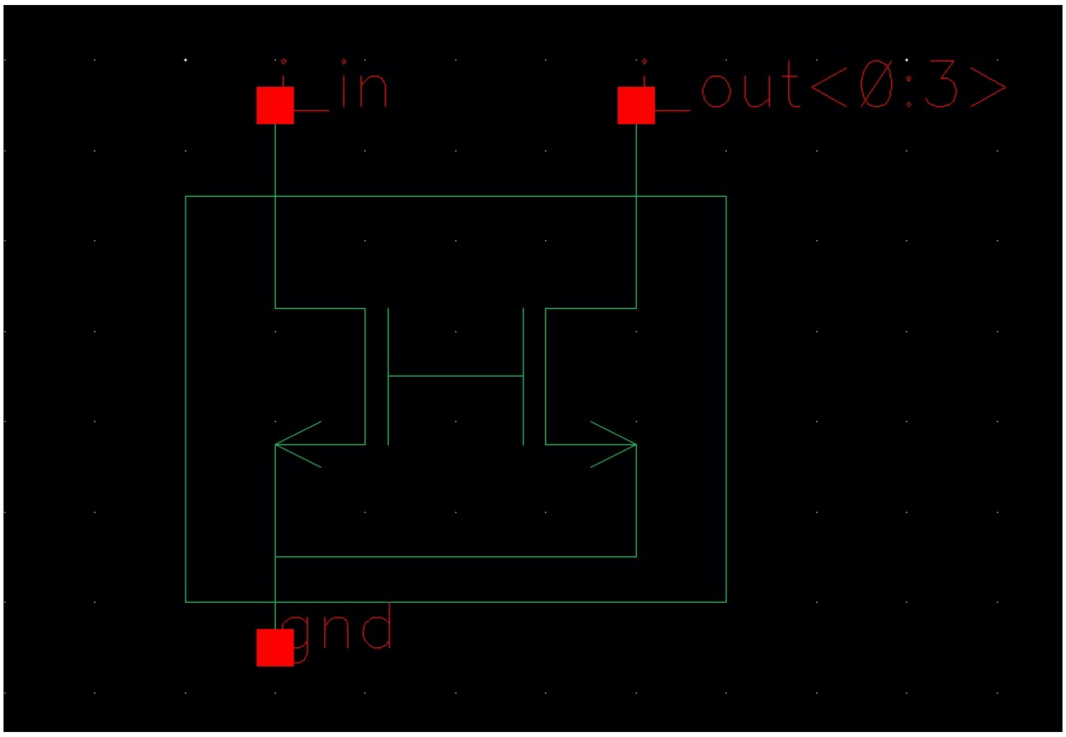

Figure 3: Result of the PCell in Symbol for Number Mirrors = 4, Input Multiplier = 4, Output Multiplier = 2, W_{in} = 1µm, L_{in} = 700nm, W_{out} = 1µm, L_{out} = 700nm, Bus = TRUE, Visible Terminals = TRUE

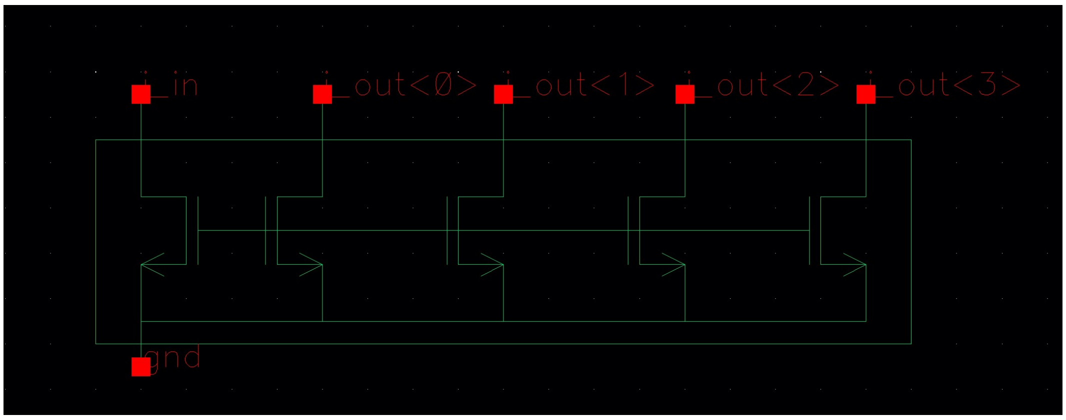

Figure 4: Result of the PCell in Symbol for Number Mirrors = 4, Input Multiplier = 4, Output Multiplier = 2, W_{in} = 1µm, L_{in} = 700nm, W_{out} = 1µm, L_{out} = 700nm, Bus = FALSE, Visible Terminals = TRUE

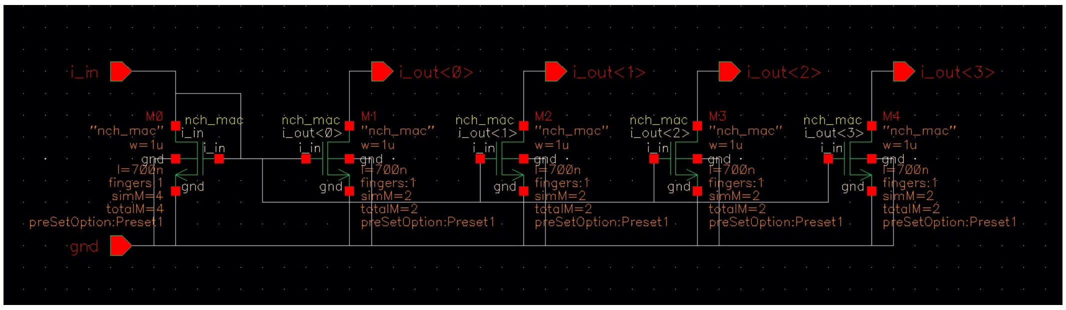

Figure 5: Result of the PCell in Schematic for Number Mirrors = 4, Input Multiplier = 4, Output Multiplier = 2, W_{in} = 1µm, L_{in} = 700nm, W_{out} = 1µm, L_{out} = 700nm

These views are DRC and LVS clean.

3.3 Simulation

Now that we have shown how the created cell views look like, we would like to show that the instances are actually working. Therefore we use the teststand package in python simulating the current mirror under different conditions. This was done using a jupyter notebook. From these simulations some of the results will be shown as plots of input versus output current.

We started by simulating a simple dc sweep analysis from which we saw that our current mirror was working as expected for hardcoded parameters. The nice thing about the parameterization of the cell is, that it is possible to change some of the instance parameters out of the simulation software allowing for a sweep over different parameters. This was for example achieved for different transistor geometries at the input and output sides. After confirming that these simulations worked correctly, we started adding MonteCarlo deviations to the transistor parameters. These simulations yielded the following results:

For an MC-Simulation of identical transistors without multipliers and 6 outputs we find.

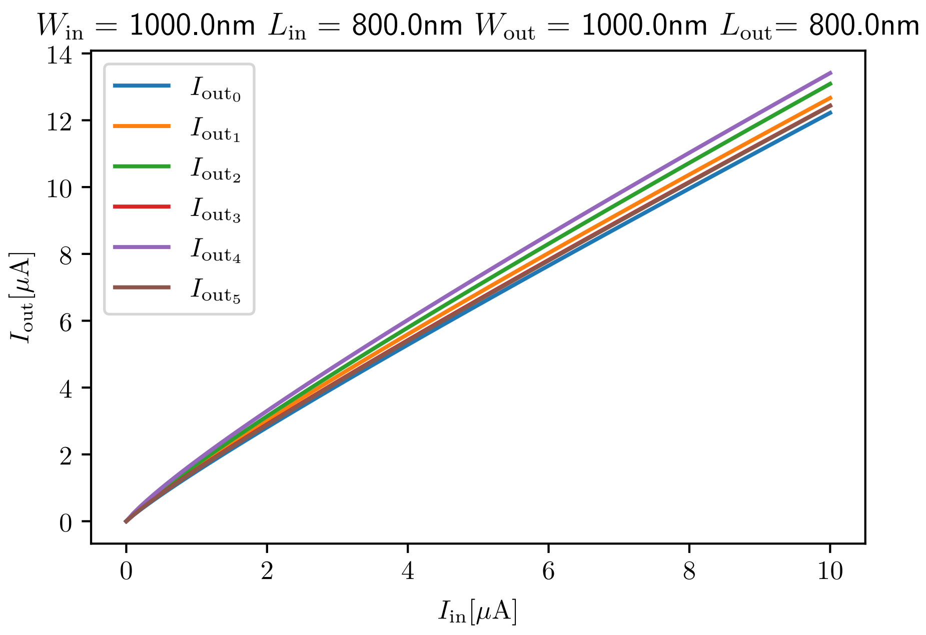

Figure 6: MC-Simulation of identical transistors without multipliers

When using a multiplier of 2 at the input and 6 at the output with otherwise identical transistors we find:

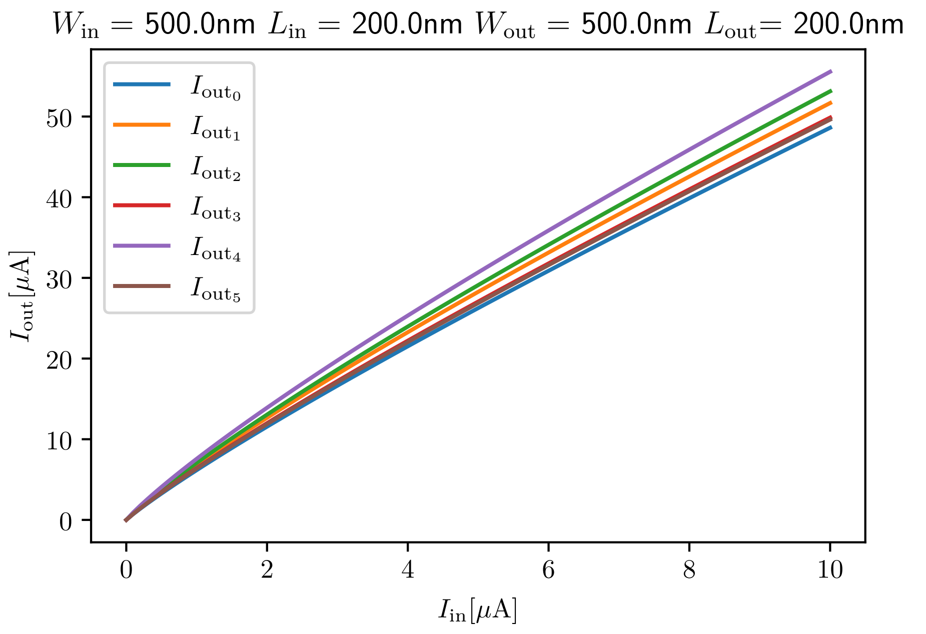

Figure 7: MC-Simulation of identical transistors with MultIn = 2 MultOut = 6

This shows that it is possible to simulate the created PCell for different parameters, and even change the parameters without interacting through the GUI directly using only the test software.

4 Summary

The aim of this manuscript was to create a documentation on how to create a PCell using the SKILL Language from Cadence®Virtuoso®. To achieve this goal the different steps one has to follow when writing the SKILL code were explained and illustrated. Then a brief explanation of the current mirror was given, followed by an example of PCell creation using the example of a current mirror. To conclude this manuscript we showed the created cellviews as well as a possible simulation of the current mirror.

5 Literature

References

Cadence Design Systems, Inc. (2002). *Custom Layout SKILL Functions Reference.* 5.0. Cadence Design Systems, Inc. 555 River Oaks Parkway, San Jose, CA 95134, USA.

Cadence Design Systems, Inc. (2004a). *Cadence Design Framework II SKILL Functions Reference.* 6.3. Cadence Design Systems, Inc. 555 River Oaks Parkway, San Jose, CA 95134, USA.

Cadence Design Systems, Inc. (2004b). *SKILL Development of Parameterized Cells.* 5.1.41. Cadence Design Systems, Inc. 555 River Oaks Parkway, San Jose, CA 95134, USA.

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

Cadence Design Systems, Inc. (2005). *Component Description Format User Guide.* 5.1.41. Cadence Design Systems, Inc. 555 River Oaks Parkway, San Jose, CA 95134, USA.

Cadence Design Systems, Inc. (2022). *Data type conversion in SKILL for numbers (integers/floats), strings, and symbols stored in various formats.* Cadence Design Systems, Inc. 555 River Oaks Parkway, San Jose, CA 95134, USA.

6 Appendix

6.1 Code Main File

```
; library/cell of PCell
library = ddGetObj("examplelib")
cell = "current_mirror"

; layout definition
PCellLayoutId = pcDefinePCell(
    list( library cell "layout" "maskLayout")

    ; parameters of PCell
    (
        ( NumberMirrors 1 )
        ( MultIn 1 )
        ( MultOut 1 )
        ( WidthIn 200n )
        ( LengthIn 60n )
        ( WidthOut 200n )
        ( LengthOut 60n)
        ( TermVisible t )
        ( Bus t )
    )

    let(()
        ; calculate number of additional fingers
        MuI = MultIn-1
        MuO = MultOut-1

        ;  set parametervalues to micrometers
        ConvFactor = 1000000
        WI = WidthIn*ConvFactor
        LI = LengthIn*ConvFactor
        WO = WidthOut*ConvFactor
```

```
LO = LengthOut*ConvFactor

; draw input transistor
DrawTransistorLayout(0.430 -0.07 WI LI nil)

; input gate connection
; metal 1
dbCreateRect(pcCellView list("M1" "drawing") list(0.15:0.435 0.32:0.72+LI))
dbCreateRect(pcCellView list("M1" "drawing") list(0.15:0.49+LI 0.43:0.72+LI))
; poly
dbCreateRect(pcCellView list("PO" "drawing") list(0.15:0.435 0.32:0.545+LI))
↪  ; input via/gate
; contact layer
NVias = int((LI-0.06)/0.21)+1
k = 0
while(k<NVias
    dbCreateRect(pcCellView list("CO" "drawing") list(0.19:0.475+k*0.21
    ↪  0.28:0.565+k*0.21))                ; input via/gate
    k++
)

; multiplier growth
i=1
while( i<=MuI
    DrawTransistorLayout(0.43+(0.2+WI)*i -0.07 WI LI t)
    dbCreateRect(pcCellView list("M1" "drawing") list(0.43:0.49+LI 0.43+(0.2+WI)*i+WI:0.72+LI)) ;
    ↪  drainconnection between transistors
    i++
)

; draw pins
DrawPinLayout(0 -0.05 "gnd")
DrawPinLayout(0.48 0.79+LI "i_in")

; iterate over mirror branches to draw output transistors
i=0
while( i<NumberMirrors

    DrawTransistorLayout(0.43+WI+(0.2+WI)*MuI+0.28+(0.28+WO+(0.2+WO)*MuO)*i -0.07 WO LO nil)
    ; multiplier growth
    j=1
    while( j<= MuO
        DrawTransistorLayout(0.43+WI+(0.2+WI)*MuI+0.28+(WO+0.2)*j+(0.28+WO+(0.2+WO)*MuO)*i -0.07
        ↪  WO LO t)
        dbCreateRect(pcCellView list("M1" "drawing")
        ↪  list(0.43+WI+(0.2+WI)*MuI+0.28+(0.28+WO+(0.2+WO)*MuO)*i:0.49+LO
        ↪  0.43+WI+(0.2+WI)*MuI+0.28+WO+(WO+0.2)*j+(0.28+WO+(0.2+WO)*MuO)*i:0.72+LO))     ;
        ↪  drainconnection between transistors
        j++
    )

    ; draw pin
    DrawPinLayout(0.43+WI+(0.2+WI)*MuI+0.28+(0.28+WO+(0.2+WO)*MuO)*i+0.05 0.79+LO strcat("i_out<"
    ↪  sprintf(nil "%d" i) ">"))
    i++
)

; PDK
if(LO<LI then
    LO = LI)
```

```
        dbCreateRect(pcCellView list("PDK" "drawing") list(0:-0.12
        ↪   0.86+WI+(0.2+WI)*MuI+(0.28+WO+(0.2+WO)*MuO)*NumberMirrors:0.89+LI))                ; Bonding
        ↪   Box
    )
)

dbSave(PCellLayoutId)
dbClose(PCellLayoutId)

; -------------------------------------------------------------------------------------------

; define the schematic
PCellSchematicId = pcDefinePCell(
    list( library cell "schematic" "schematic")

    ; parameters of PCell
    (
        ( NumberMirrors 1 )
        ( MultIn 1 )
        ( MultOut 1 )
        ( WidthIn "200n" )
        ( LengthIn "60n" )
        ( WidthOut "200n" )
        ( LengthOut "60n" )
        ( TermVisible t )
        ( Bus t )
    )

    let(()
        InputFETInst = InsertFET(0 0 "MY" WidthIn LengthIn MultIn)

        ; create the gnd net
        gnd = dbCreateNet(pcCellView "gnd")
        gnd~>sigType = "ground"        ; change signaltype off the net

        ; add wires to net
        ; create ground connection
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.25:-0.1875 -0.25:-0.5)) gnd)
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.5:-0.5 -0.25:-0.5)) gnd)

        ; create bulk connection
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.25:0 -0.375:0)) gnd)
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.375:0 -0.375:-0.5)) gnd)

        ; connect instace terminals to net
        dbCreateConn(gnd InputFETInst dbFindTermByName(NMOSMaster "S"))
        dbCreateConn(gnd InputFETInst dbFindTermByName(NMOSMaster "B"))

        ; create the i_in net
        i_in = dbCreateNet(pcCellView "i_in")
        ; signal type is "signal", doesn't need changing

        ; add wires to net
        ; create input connection
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.25:0.1875 -0.25:0.5)) i_in)
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.5:0.5 -0.25:0.5)) i_in)

        ; connect instace terminals to net
        dbCreateConn(i_in InputFETInst dbFindTermByName(NMOSMaster "D"))
        dbCreateConn(i_in InputFETInst dbFindTermByName(NMOSMaster "G"))
```

```
; create output branches according to parameter
i=0
while( i<NumberMirrors

    ; instantiate mirror transistor
    OutputFETInst = InsertFET(0.5+1*i 0 "R0" WidthOut LengthOut MultOut)

    ; create ground connection
    dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.25+1*i:-0.5
    ↪  0.75+1*i:-0.5)) gnd)
    dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( 0.75+1*i:-0.5
    ↪  0.75+1*i:-0.1875)) gnd)

    ; connect instace terminals to net
    dbCreateConn(gnd OutputFETInst dbFindTermByName(NMOSMaster "S"))

    ; create the out net
    ; create output net names, each net different name according to parameter
    ↪  (i_out<0:NumberMirrors>)
    i_out = dbCreateNet(pcCellView strcat("i_out<" sprintf(nil "%d" i) ">"))

    ; connect instace terminals to net
    ;dbCreateConn(i_out OutputFETInst dbFindTermByName(NMOSMaster "D"))  ; both possible
    dbCreateConnByName(i_out OutputFETInst "D")

    ; add wires to net
    ; create output connection
    dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list(0.75+1*i:0.1875
    ↪  0.75+1*i:0.5)) i_out)
    dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list(0.75+1*i:0.5 0.875+1*i:0.5))
    ↪  i_out)

    ; create out pin
    pinMaster_o = dbOpenCellViewByType("basic" "opin" "symbol")                ; load pin
    term = dbCreateTerm(i_out "" "output")                                                     ;
    ↪  create terminal to connect to
    i_out = dbCreatePin(i_out dbCreateInst( pcCellView pinMaster_o "" 0.875+1*i:0.5 "R0") "")

    ; create ground and bulk connection
    dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( 0.75+1*i:0 0.875+1*i:0))
    ↪  gnd)
    dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( 0.875+1*i:0
    ↪  0.875+1*i:-0.5)) gnd)
    dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( 0.875+1*i:-0.5
    ↪  0.75+1*i:-0.5)) gnd)

    ; connect instace terminals to net
    dbCreateConn(gnd OutputFETInst dbFindTermByName(NMOSMaster "B"))

    ; create connection between gates
    ; first mirror
    if(i == 0 then
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( 0:0 0.5:0)) i_in)
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( -0.25:0.375
        ↪  0.125:0.375)) i_in)
        dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list( 0.125:0.375 0.125:0))
        ↪  i_in)
    )

    ; > first mirror
    if(i != 0 then
                i = i-1
                dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list(0.25:0 0.25:-0.25))
                ↪  i_in)
                dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list(0.25+1*i:-0.25
                ↪  1.5+1*i:-0.25)) i_in)
                dbAddFigToNet(dbCreateLine( pcCellView list("wire" "drawing") list(1.5+1*i:-0.25
                ↪  1.5+1*i:0)) i_in)
                i=i+1
            )

            ; connect instace terminals to net
            dbCreateConn(i_in OutputFETInst dbFindTermByName(NMOSMaster "G"))

            i++
        )

        ; load input pin for use
        pinMaster_i = dbOpenCellViewByType("basic" "ipin" "symbol")                ; load pin
        ; connect input pin to net
        term = dbCreateTerm(i_in "i_in" "input")                                          ; create
        ↪  terminal to connect to
        i_in = dbCreatePin(i_in dbCreateInst( pcCellView pinMaster_i "" -0.5:0.5 "R0") "i_in")

        ; connect ground pin to net
        term = dbCreateTerm(gnd "gnd" "input")
        gnd_pin = dbCreatePin(gnd dbCreateInst( pcCellView pinMaster_i "" -0.5:-0.5 "R0") "gnd_pin")
    )
)

dbSave(PCellSchematicId)
dbClose(PCellSchematicId)

; ---------------------------------------------------------------------------------------------

; define the symbol
PCellSymbolId = pcDefinePCell(
    list( library cell "symbol" "schematicSymbol")
    (
        ( NumberMirrors 1 )
        ( MultIn 1 )
        ( MultOut 1 )
        ( WidthIn "200n" )
        ( LengthIn "60n" )
        ( WidthOut "200n" )
        ( LengthOut "60n" )
        ( TermVisible t )
        ( Bus t )
    )
    let(()
        ; load the symbols for later instantiation
        InputFET = dbOpenCellViewByType( "examplelib" "current_mirror" "symbol_in")
        OutputFET = dbOpenCellViewByType( "examplelib" "current_mirror" "symbol_out")

        ; create input
        ; create input transistor
        InputFETInst = dbCreateParamInst(pcCellView InputFET nil 0:0 "R0")

        ; create the gnd net
        gnd = dbCreateNet(pcCellView "gnd")
        gnd~>sigType = "ground" ; change signaltype of the net
```

```
; create the i_in Net
i_in = dbCreateNet(pcCellView "i_in")
; signal type is "signal", doesn't need changing

; create terminals
; load master pin according to TermVisible
if(TermVisible == t then
    PinMaster = dbOpenCellViewByType("basic" "sympin" "symbol")
)

if(TermVisible != t then
    PinMaster = dbOpenCellViewByType("basic" "sympin" "symbolNN")
)

; setting pin parameters (overbar, justification)
PinMaster~>shapes~>isOverbar = nil
PinMaster~>shapes~>justify = "centerLeft"

; connect terminals to net
; i_in terminal
term = dbCreateTerm(i_in "i_in" "input")
i_in = dbCreatePin(i_in dbCreateInst( pcCellView PinMaster "" 0:0.375 "R0") "i_in")

; gnd terminal
term = dbCreateTerm(gnd "gnd" "input")
gnd_pin = dbCreatePin(gnd dbCreateInst( pcCellView PinMaster "" 0:-0.375 "R0") "gnd_pin")


;create output
; create output branches according to parameters
; create output bus
if(Bus == t then

    ; create output transistor
    OutputFETInst = dbCreateParamInst(pcCellView OutputFET nil 0.5:0 "R0")

    ; add ground connection
    dbCreateLine( pcCellView list(231 "drawing") list( 0.5:-0.25 0:-0.25))

    ; create output connection with bus naming (i_out<0:NumberMirrors>)
    i_out = dbCreateNet(pcCellView strcat("i_out<0:" sprintf(nil "%d" NumberMirrors-1) ">"))

    ; create output terminal and pin
    term = dbCreateTerm(i_out "" "output")
    i_out = dbCreatePin(i_out dbCreateInst( pcCellView PinMaster "" 0.5:0.375 "R0") "")

    ; create draw box
    dbCreateRect(pcCellView list(231 "drawing") list(-0.125:-0.3125 0.125+0.5:0.25))
    ; create selection box
    dbCreateRect(pcCellView list(236 "drawing") list(-0.25:-0.375 0.25+0.5:0.375))
)

; create single outputs
if(Bus != t then

    ; create output branches according to parameter
    i=0
    while( i<NumberMirrors
        ; create output transistor
        OutputFETInst = dbCreateParamInst(pcCellView OutputFET nil 0.5+0.5*i:0 "R0")   ; next one
        ↪  +0.5
```

```
                ; add ground connection
                dbCreateLine( pcCellView list(231 "drawing") list( 0.5+0.5*i:-0.25 0+0.5*i:-0.25))

                ; create the out net
                ; create output net names, each net different name according to parameter
                ↪  (i_out<0:NumberMirrors>)
                i_out = dbCreateNet(pcCellView strcat("i_out<" sprintf(nil "%d" i) ">"))

                ; create output terminal and pin
                term = dbCreateTerm(i_out "" "output")
                i_out = dbCreatePin(i_out dbCreateInst( pcCellView PinMaster "" 0.5+0.5*i:0.375 "R0") "")

                ; create gate connection for instances > 1
                if(i != 0 then
                    i = i-1
                    dbCreateLine( pcCellView list(231 "drawing") list(0.3125+0.5*i:0 0.75+0.5*i:0))
                    i=i+1
                )
                i++
            )

            ; create draw box
            dbCreateRect(pcCellView list(231 "drawing") list(-0.125:-0.3125 0.125+0.5*i:0.25))
            ; create selection box
            dbCreateRect(pcCellView list(236 "drawing") list(-0.25:-0.375 0.25+0.5*i:0.375))
        )
    )
)

dbSave(PCellSymbolId)
dbClose(PCellSymbolId)

; ------------------------------------------------------------------------------------------

;  definition of CDF parameters
let(
    ( cellId cdfId )                                                              ; searching the cell
    ↪  name
    unless( cellId = ddGetObj( library~>name cell )
        error( "Could not find cell %s." cell )                  ; error, if cell not in library
    )
    when( cdfId = cdfGetBaseCellCDF( cellId )
        cdfDeleteCDF( cdfId )
    )
    cdfId = cdfCreateBaseCellCDF( cellId )                        ; create standard cell

    ; Create settable parameters
    ; number of output mirrors
    cdfCreateParam( cdfId
        ?name "NumberMirrors"
        ?type "int"
        ?prompt "How Many Mirrors"
        ?defValue 1
        ?callback "if( cdfgData->NumberMirrors~>value < 1 then
            cdfgData~>NumberMirrors~>value = 1)"
    )

    ; multiplier in
    cdfCreateParam( cdfId
        ?name "MultIn"
    ?type "int"
    ?prompt "Input Transistor Multiplier"
    ?defValue 1
    ?callback "if( cdfgData~>MultIn~>value < 1 then
        cdfgData~>MultIn~>value = 1)"
)

; multiplier out
cdfCreateParam( cdfId
    ?name "MultOut"
    ?type "int"
    ?prompt "Output Transistor Multiplier"
    ?defValue 1
    ?callback "if( cdfgData~>MultOut~>value < 1 then
        cdfgData~>MultOut~>value = 1)"
)

; input width
cdfCreateParam( cdfId
    ?name "WidthIn"
    ?type "string"
    ?prompt "Input Transistor Width"
    ?defValue "200n"
    ?parseAsCEL "yes"
    ?parseAsNumber "yes"
    ?units "lengthMetric"
)

; input length
cdfCreateParam( cdfId
    ?name "LengthIn"
    ?type "string"
    ?prompt "Input Transistor Length"
    ?defValue "60n"
    ?parseAsCEL "yes"
    ?parseAsNumber "yes"
    ?units "lengthMetric"
)

; output width
cdfCreateParam( cdfId
    ?name "WidthOut"
    ?type "string"
    ?prompt "Output Transistor Width"
    ?defValue "200n"
    ?parseAsCEL "yes"
    ?parseAsNumber "yes"
    ?units "lengthMetric"
)

; output length
cdfCreateParam( cdfId
    ?name "LengthOut"
    ?type "string"
    ?prompt "Output Transistor Length"
    ?defValue "60n"
    ?parseAsCEL "yes"
    ?parseAsNumber "yes"
    ?units "lengthMetric"
)
```

```
    ; Visible terminal names
    cdfCreateParam( cdfId
        ?name "TermVisible"
        ?type "boolean"
        ?prompt "Visible Terminals"
        ?defValue t
    )

    ; output bus
    cdfCreateParam( cdfId
        ?name "Bus"
        ?type "boolean"
        ?prompt "Make output Bus"
        ?defValue t
    )

    ;;; Simulator Information
    cdfId->simInfo = list( nil )
    cdfId->simInfo->UltraSim = '( nil )
    cdfId->simInfo->ams = '( nil )
    cdfId->simInfo->auCdl = '( nil )
    cdfId->simInfo->auLvs = '( nil )
    cdfId->simInfo->cdsSpice = '( nil )
    cdfId->simInfo->hspiceD = '( nil )
    cdfId->simInfo->hspiceS = '( nil )
    cdfId->simInfo->spectre = '( nil )
    cdfId->simInfo->spectreS = '( nil )

    cdfSaveCDF(cdfId)
)
```

6.2 Code Helper File

```
; function definitions
↪ -----------------------------------------------------------------------------------------------
; layout
procedure( DrawTransistorLayout(x y W L m)

    dbCreateRect(pcCellView list("M1" "drawing") list(x-0.43:y x+0.43+W:y+0.14))
    ↪  ; gnd line
    dbCreateRect(pcCellView list("M1" "drawing") list(x:y x+W:y+0.45))                          ; source
    ↪  connection transistor
    if(m!=t then
        dbCreateRect(pcCellView list("M1" "drawing") list(x:y+0.56+L
        ↪  x+W:y+0.96+L))                                ; drain connection transistor
    )

    dbCreateRect(pcCellView list("OD" "drawing") list(x:y+0.32 x+W:y+0.69+L))                   ;
    ↪  transistor
    dbCreateRect(pcCellView list("OD" "drawing") list(x-0.145:y+0.01 x+0.145+W:y+0.13))         ; gnd
    ↪  connection rail / bulk

    NVias = int((W-0.2)/0.21)+1
    k = 0
    while(k<NVias
        dbCreateRect(pcCellView list("CO" "drawing") list(x+0.055+k*0.21:y+0.56+L
        ↪  x+0.145+k*0.21:y+0.65+L))           ; input transistor vias
        dbCreateRect(pcCellView list("CO" "drawing") list(x+0.055+k*0.21:y+0.36
        ↪  x+0.145+k*0.21:y+0.45))                       ; gnd transistor vias
        dbCreateRect(pcCellView list("CO" "drawing") list(x+0.055+k*0.21:y+0.025
        ↪  x+0.145+k*0.21:y+0.115))                ; gnd connection rail vias / bulk
        k++
    )

    dbCreateRect(pcCellView list("PO" "drawing") list(x-0.14:y+0.505
    ↪  x+W+0.14:y+0.505+L))                                         ; gate

    ; doping
    dbCreateRect(pcCellView list("NP" "drawing") list(x-0.43:y+0.19
    ↪  x+0.43+W:y+0.96+L))                         ; NP region
    dbCreateRect(pcCellView list("PP" "drawing") list(x-0.43:y-0.05
    ↪  x+0.43+W:y+0.19))                         ; PP region

)

procedure( DrawPinLayout(x y name)

    dbCreateRect(pcCellView list("M1" "pin") list(x:y x+0.1:y+0.1))
    dbCreateLabel(pcCellView list("M1" "pin") x+0.05:y+0.05 name "centerCenter" "R0" "stick" 0.05)

)

; -------------------------------------------------------------------------------------------

; schematic
procedure( InsertFET(X Y Orientation W L M)
    ; load the transistor for later instantiation
    NMOSMaster = dbOpenCellViewByType("gpdk" "nch_mac" "symbol")

    ; create start transistor
    Inst = dbCreateParamInst(pcCellView NMOSMaster nil X:Y Orientation)

    ; set multiplier of transistor
    cdfgData=cdfGetInstCDF(Inst)
    cdfgData~>simM~>value=sprintf(nil "%d" M)
    cdfgData~>totalM~>value=sprintf(nil "%d" M)

    ; set channel geometry of transistor
    cdfgData~>w~>value = W
    cdfgData~>l~>value = L

    Inst
)
; -------------------------------------------------------------------------------------------
```

6.3 Code LibInit File

```
load("/hyperfast/home/khaas/scratch/pcells/helpers.il")
if(isCallable('hiSetBindKey)    ; check for need of graphic interface because netlistproblem with ocean
    load("/hyperfast/home/khaas/scratch/pcells/current_mirror.il")
)
```

6.4 Useful forum links

Further sources of information were the Cadence®Community forum `https://community.cadence.com/cadence_technology_forums/f`, with special thanks to Andrew Beckett for answering nearly every question there, as well as the Cadence®support website helping with things like data conversion `https://support1.cadence.com/public/docs/content/11787986.html`.

6.5 Thanks

Special thanks goes to Sebastian Billaudelle for helping with questions according the content, as well as the appearance of this report. Further, I would like to thank Philipp Dauer, for also helping with the content and implementation questions. And finally thanks to Heike Schlatterer for helping with language and grammar questions.