

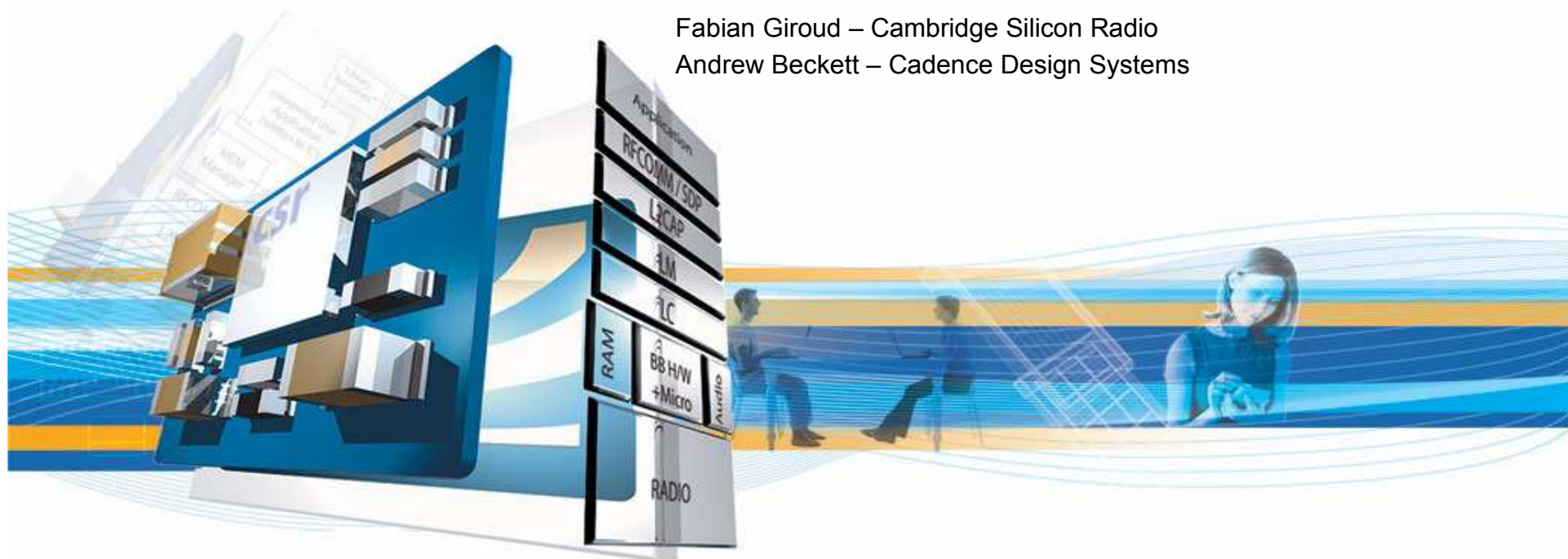


Changing the way the world connects

## Automated Creation of a Customized PDK for 90nm

---

Fabian Giroud – Cambridge Silicon Radio  
Andrew Beckett – Cadence Design Systems





## Objectives

---

- Development of a customized PDK based on a standard 90nm foundry PDK, using an automated process.
- Introduction of the PDK and flow to a team new to Cadence tools
  - Take advantage of existing practices, whilst adopting improvements that new tools offer
- Work was done in partnership with Cadence Design Systems



## Why are we customizing the PDK?

- Want a PDK with component names and parameters independent of technology
  - Eases migration of designs to other technologies
- Additional capabilities not available in the standard PDK
  - For example:
    - Double contacts on poly gates, even for minimum size device
    - Ensuring pins are on routing grid
  - Improvement of yield and consistency of design
  - Some component types (e.g. I/O devices) not available, so need to be added
- Simplification of parameters
  - Too much flexibility makes review harder, takes longer for the designer, and makes consistency of methodology more difficult
- Support for enhanced device simulation models
- Freedom to make local improvements in a timely fashion



## Principles for custom PDK development

---

- Don't use CDF callbacks for deriving data
  - OK to use callbacks for range checking
  - OK to use for calculating estimated values to display to user
- Want to be able to build PDK automatically from a mapping file
  - Allows addition of new device variants easily
- Use the actual pcells in the source PDK and add customisations around them
  - Avoids unnecessary rework and QA
- Automated generation of testbenches
- Must support schematic driven layout with VirtuosoXL



## Why CDF callbacks can be dangerous (part 1)

- Problems come from using CDF callbacks to derive values which are then used in simulation and to define pcells
- With such a “push” model, if the callback doesn’t get called, you have inconsistent device parameters
  - Imagine a transistor where you enter *total\_width* and *nfingers*
  - A CDF callback generates a new parameter *finger\_width* which is used to drive the pcell
  - Simulation uses *total\_width*
  - LVS uses *finger\_width* and *nfingers* for comparison.
  - The *total\_width* was changed by somebody using SKILL
    - `geGetSelSet()~>total_width="1u"`
  - The simulation now will be using one dimension, and the layout and LVS using another – your design does not match what you simulated, but LVS is clean!



## Why CDF callbacks can be dangerous (part 2)

- CDF callbacks prevent parameterization of devices
  - Imagine an inverter where the *width* of one of the transistors is set to a hierarchical parameter  $pPar("inv\_width")$
  - You have two instances of the inverter, one with *inv\_width* set to  $1u$  and one set to  $2u$ .
  - There is a CDF callback on the *width* parameter on the transistor which derives another parameter *sim\_width* which is used for simulation
  - CDF callbacks are form field callbacks and so get called when the value of *width* is changed.
  - Even if the value of  $pPar("inv\_width")$  can be found using the hierarchical path that is in place at the time the transistor inside the inverter is edited...
    - This only applies to that occurrence – other occurrences will be incorrect
    - What happens if the value of *inv\_width* is later changed?
- Essentially, the derivation of another parameter can only safely be done at “elaboration” time in the simulator or when layout generated.





## So what's the solution for CDF Callbacks?

- Rather than deriving values in CDF callbacks, use a “pull” model
  - Calculate values in the simulation models
  - Calculate values in the pcells
  - Simulation and pcells are driven from the *entered* values
- But designer still wants feedback of some derived values
  - For example, drain and source area/perimeter
  - Can do calculation (if possible – i.e. if the devices have not been parameterised) and display the *estimated* values.
  - If *estimated* values get out of sync, there is no danger, since simulation and layout are based on the *entered* values.
- What if I need the same equation in both simulation and pcell – how can I ensure they are consistent?
  - Using careful checking, or using a macro language such as *M4*



## Modification of symbols used in PDK

---

- The symbols used in the CSR PDK are based on those in the source PDK
- Modifications are done to the graphics of the symbol to distinguish between different transistor types
  - E.g. Low leakage, RF, etc have slightly different graphics
  - Modification is done from the base symbol using SKILL
- Pin sizes on components are made smaller for cosmetic reasons
  - Puts focus on important parameters and improves speed of review
- Additional interpreted labels for displaying some custom CDF parameters





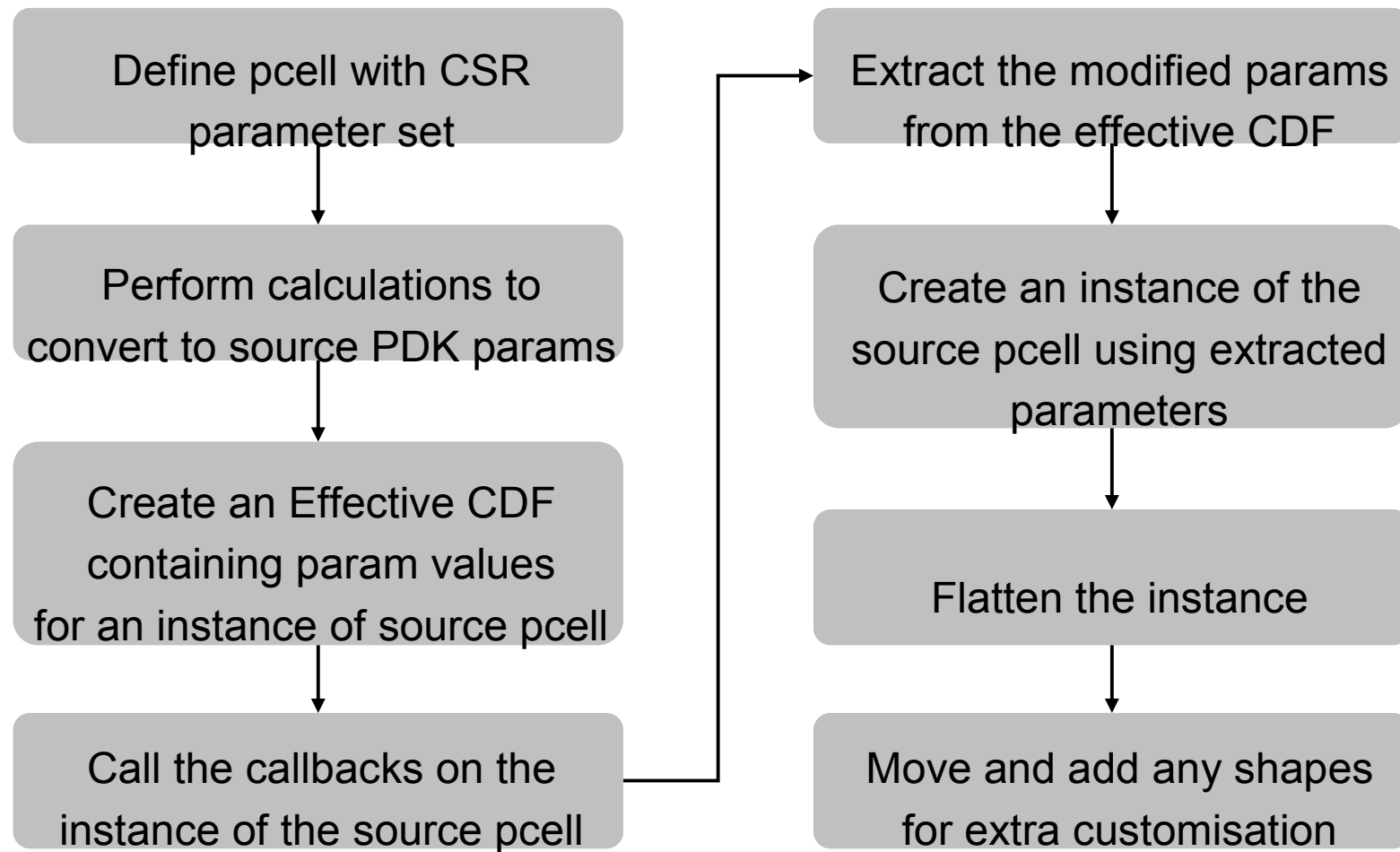
## Building custom Pcells

---

- Pcells (Parameterised Cells) are being created for each device using a “wrapper” methodology
- Effectively this uses as much as possible of the underlying source Pcell without needing to reinvent the wheel
- Wrappers can be common to a whole family of device types – e.g. all MOSFETs can share the same wrapper code
- Allows transformation of desired parameters to underlying parameters
- Allows calculations previously done in callbacks to be performed in the Pcell
- Allows additional creation and adjustment of shapes, connectivity etc in the wrapper



## Pcell wrapper flow





## Testing

---

- For each class of component, build a schematic with an instance of each type of component, connected to pins
- Create a simulation testbench with parameterised sources connected to each device pin
- Use OCEAN to run simulations
  - Also use OCEAN to validate device models
- Invoke CDF callbacks on all instances in schematic
  - Checks that callbacks for estimates do not break with SKILL errors
- Automatically generate layout using VirtuosoXL SKILL functions
  - Place pins over device pins
  - Add substrate/well contacts where needed
- Automatically run DRC and LVS



## Conclusions

---

- Benefits
  - Robust PDK, simple to use
  - 2-3 months of PDK development and testing time with extensive feedback from end users to guide development
  - A complete build of the PDK from scratch can be done in a few minutes, since the whole process is automated.
  - Same methodology and code can be rapidly applied to another PDK
    - 65nm PDK in ~ 1 week!
- Concerns
  - Some dependency on consistency of delivered source PDK
  - In practice most things can be corrected with a small amount of work
- Future work
  - More extensive automated testing
  - Generation of large number of variants in order to do testing of the CDF callbacks used for range checking and estimated values – and also to check the pcells under many more conditions