Composing Modular Flow Generators with Python-Based Static Checking to Enable Agile Principles in Physical Design

Stanford University

October 14, 2021

Christopher Torng, ctormg@stanford.edu
Building complex systems in advanced technologies is challenging

Challenges across the stack

- Software stack
- Logical hardware design
- Physical hardware design

Majority of effort today is not on the "secret sauce" ... true for both the design and the flow that constructs it

To productively build these systems, we need to somehow extract very high code reuse

But reuse is challenging, especially in physical design

- "One big release" model
The "One Big Release" Model of Hardware Design

A model where high stakes and strict schedules lead to heavy customization.

1. Scripts tied to a particular technology (process node, library vendors, pdk)

2. Scripts tied to a particular design (language, custom power, assumptions)
Modern Code Reuse Solutions in Physical Design

Tcl templates and generators with parameters

- Create initial technology-agnostic and design-agnostic Tcl flows
- Examples of generic flows
  - UCB Hammer
  - Synopsys Reference Methodology
  - Cadence Innovus Foundation Flow

Problems

- These Tcl flows enable efficient reuse until a need arises for which no parameter exists
- Modern machine learning CAD solutions may be not written in Tcl in the first place, but must still compose with existing flows
- Reuse of generic code is great, but most design effort goes into custom code, which is challenging to reuse
Modular Flow Generators with Flow Consistency and Instrumentation Layer

1. Modular nodes are composed into flows
2. FCI layer executes static checks for consistency
3. Nodes are instrumented for agile principles

Generic Nodes (Reusable)
Custom Nodes (Reusable)
Custom Nodes (Not Reusable)

Modular Flow Generator
Flow Consistency and Instrumentation (FCI) Layer
Schema for the Modular Node Abstraction

Modular Node Configuration Schema

<table>
<thead>
<tr>
<th>YAML</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name: &lt;string&gt;</td>
<td></td>
</tr>
<tr>
<td>inputs:</td>
<td></td>
</tr>
<tr>
<td>- &lt;string&gt;</td>
<td></td>
</tr>
<tr>
<td>outputs:</td>
<td></td>
</tr>
<tr>
<td>- &lt;string&gt;</td>
<td></td>
</tr>
<tr>
<td>commands:</td>
<td></td>
</tr>
<tr>
<td>- &lt;string&gt;</td>
<td></td>
</tr>
<tr>
<td>parameters:</td>
<td></td>
</tr>
<tr>
<td>- &lt;string&gt;: &lt;string&gt;</td>
<td></td>
</tr>
</tbody>
</table>

Modular nodes are *file-aware functions* and can capture different tasks

- Bump routing methodology for flip chip
- Approach for design for manufacture (DFM) structures
- Adding power domains
- ECO timing fixes

**Important**: Abstraction must be sufficiently flexible for:

- Large code blocks
- Fine-grain code blocks (glue script steps)
- Different commercial EDA vendors
- ML CAD tools
- Open-source tools.
Schema for the Modular Node Abstraction

Secondary Schema for the Modular Node Abstraction

**Modular Node Configuration Schema**

- **YAML**
  - name: <string>
  - inputs:
    - <string>
  - outputs:
    - <string>
  - commands:
    - <string>
  - parameters:
    - <string>: <string>

**Graph Visualization**

- tech
- design.v
- synthesis
- design.v

**Modular nodes are file-aware functions** and can capture different tasks:

- Bump routing methodology for flip chip
- Approach for design for manufacture (DFM) structures
- Adding power domains
- ECO timing fixes

**Important**: Abstraction must be sufficiently flexible for:

- Large code blocks
- Fine-grain code blocks (glue script steps)
- Different commercial EDA vendors
- ML CAD tools
- Open-source tools.
Schema for the Modular Node Abstraction

Modular nodes are file-aware functions and can capture different tasks:

- Bump routing methodology for flip chip
- Approach for design for manufacture (DFM) structures
- Adding power domains
- ECO timing fixes

Important: Abstraction must be sufficiently flexible for:

- Large code blocks
- Fine-grain code blocks (glue script steps)
- Different commercial EDA vendors
- ML CAD tools
- Open-source tools.
Three Classes of Reusable Code in Physical Design

Straightforward, given a modular abstraction
### Three Classes of Reusable Code in Physical Design

<table>
<thead>
<tr>
<th>Base Tool</th>
<th>#</th>
<th>Description of Modular Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadence Genus</td>
<td>2</td>
<td>Synthesis, generate post-pnr lib</td>
</tr>
<tr>
<td>Cadence Innovus</td>
<td>14</td>
<td>Init, place, cts, route, postroute, signoff, post-pnr eco design, foundation flow setup, hold-fixing nodes, power grid setup</td>
</tr>
<tr>
<td>Cadence Pegasus</td>
<td>4</td>
<td>DRC, LVS, GDS merging, metal fill</td>
</tr>
</tbody>
</table>
Three Classes of Reusable Code in Physical Design

<table>
<thead>
<tr>
<th>Base Tool</th>
<th>#</th>
<th>Description of Modular Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadence Genus</td>
<td>2</td>
<td>Synthesis, generate post-pnr lib</td>
</tr>
<tr>
<td>Cadence Innovus</td>
<td>14</td>
<td>Init, place, cts, route, postroute, signoff, post-pnr eco design, foundation flow setup, hold-fixing nodes, power grid setup</td>
</tr>
<tr>
<td>Cadence Pegasus</td>
<td>4</td>
<td>DRC, LVS, GDS merging, metal fill</td>
</tr>
<tr>
<td>Synopsys DC</td>
<td>1</td>
<td>Synthesis</td>
</tr>
<tr>
<td>Synopsys Formality</td>
<td>1</td>
<td>Logical equivalence check</td>
</tr>
<tr>
<td>Synopsys PT(PX)</td>
<td>6</td>
<td>Timing/power signoff, ECO flow, gen lib/db, RTL- and gate-level power estimation</td>
</tr>
<tr>
<td>Synopsys VCS</td>
<td>2</td>
<td>RTL- and gate-level simulation, vcd2saif</td>
</tr>
<tr>
<td>Mentor Calibre</td>
<td>7</td>
<td>DRC, LVS, GDS merging, metal fill, converting verilog2spice, drawing chip art</td>
</tr>
</tbody>
</table>
### Three Classes of Reusable Code in Physical Design

<table>
<thead>
<tr>
<th>Base Tool</th>
<th>#</th>
<th>Description of Modular Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadence Genus</td>
<td>2</td>
<td>Synthesis, generate post-pnr lib</td>
</tr>
<tr>
<td>Cadence Innovus</td>
<td>14</td>
<td>Init, place, cts, route, postroute, signoff, post-pnr eco design, foundation flow setup, hold-fixing nodes, power grid setup</td>
</tr>
<tr>
<td>Cadence Pegasus</td>
<td>4</td>
<td>DRC, LVS, GDS merging, metal fill</td>
</tr>
<tr>
<td>Synopsys DC</td>
<td>1</td>
<td>Synthesis</td>
</tr>
<tr>
<td>Synopsys Formality</td>
<td>1</td>
<td>Logical equivalence check</td>
</tr>
<tr>
<td>Synopsys PT(PX)</td>
<td>6</td>
<td>Timing/power signoff, ECO flow, gen lib/db, RTL- and gate-level power estimation</td>
</tr>
<tr>
<td>Synopsys VCS</td>
<td>2</td>
<td>RTL- and gate-level simulation, vcd2saif</td>
</tr>
<tr>
<td>Mentor Calibre</td>
<td>7</td>
<td>DRC, LVS, GDS merging, metal fill, converting verilog2spice, drawing chip art</td>
</tr>
<tr>
<td>Open-Source</td>
<td>8</td>
<td>Synthesis (yosys), place (graywolf), place (RePlAce [10]), route (qroutert) LVS (netgen), DRC (magic) gds2spice and def2spice (magic)</td>
</tr>
</tbody>
</table>

| Total # of Nodes | 45 |

---

Christopher Torng  -  Composing Modular Flow Generators with Python-Based Static Checking to Enable Agile Principles in Physical Design
### Three Classes of Reusable Code in Physical Design

<table>
<thead>
<tr>
<th>Base Tool</th>
<th>#</th>
<th>Description of Modular Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadence Genus</td>
<td>2</td>
<td>Synthesis, generate post-pnr lib</td>
</tr>
<tr>
<td>Cadence Innovus</td>
<td>14</td>
<td>Init, place, cts, route, postroute, signoff, post-pnr eco design, foundation flow setup, hold-fixing nodes, power grid setup</td>
</tr>
<tr>
<td>Cadence Pegasus</td>
<td>4</td>
<td>DRC, LVS, GDS merging, metal fill</td>
</tr>
<tr>
<td>Synopsys DC</td>
<td>1</td>
<td>Synthesis</td>
</tr>
<tr>
<td>Synopsys Formality</td>
<td>1</td>
<td>Logical equivalence check</td>
</tr>
<tr>
<td>Synopsys PT(PX)</td>
<td>6</td>
<td>Timing/power signoff, ECO flow, gen lib/db, RTL- and gate-level power estimation</td>
</tr>
<tr>
<td>Synopsys VCS</td>
<td>2</td>
<td>RTL- and gate-level simulation, vcd2saif</td>
</tr>
<tr>
<td>Mentor Calibre</td>
<td>7</td>
<td>DRC, LVS, GDS merging, metal fill, converting verilog2spice, drawing chip art</td>
</tr>
<tr>
<td>Open-Source</td>
<td>8</td>
<td>Synthesis (yosys), place (graywolf), place (RePlAce [10]), route (qrouter) LVS (netgen), DRC (magic) gds2spice and def2spice (magic)</td>
</tr>
<tr>
<td>Total # of Nodes</td>
<td>45</td>
<td></td>
</tr>
</tbody>
</table>

**Challenge:** How to reuse code in these two classes?

**Open-Source Technologies**
- SkyWater 130nm [11], FreePDK45 with NanGate Open Cell Library
Static property checks to enable reuse of customized code

set offset 7.1
set aon_width 14
set aon_height 11

set polypitch_y [dbGet top.fPlan.coreSite.size_y]
set polypitch_x [dbGet top.fPlan.coreSite.size_x]

set aon_height_snap [expr ceil($aon_height/$polypitch_y)*$polypitch_y]
set aon_lx [expr $width * 0.15 - $aon_width/2 + $offset - 0.18]
set aon_lx_snap [expr ceil($aon_lx/$polypitch_x)*$polypitch_x]
set aon ux [expr $width * 0.15 + $aon_width/2 + $offset - 3]
set aon_ux_snap [expr ceil($aon ux/$polypitch_x)*$polypitch_x]

modifyPowerDomainAttr AON
  -box $aon_lx_snap [expr $height - $aon_height_snap - 10*$polypitch_y]
  $aon_ux_snap [expr $height - 10*$polypitch_y]
  -minGaps $polypitch_y $polypitch_y [expr $polypitch_x*6] [expr $polypitch_x*6]
Static property checks to enable reuse of customized code

Design Intent Block

```tcl
proc mflowgen.intent.<string> {<list: string>} {
    <string>  \ Block Name  Args
    array set mflowgen.property.<string> {
        property <boolean expr>  \ Property Name
        describe <string>  \ Property Expression
    }
    (...)
    Property Message
}
```

Implementation Block

```tcl
proc mflowgen.implement.<string> {} {
    <string>
    return [ list <list:string> ]
}
```

- Separate design intent from implementation
- Static analysis extracts/verifies properties
- One intent block can pair with multiple different implementations

Composing design intent and implementation

```tcl
mflowgen.intent.<string> { } [ mflowgen.implement.<string> ]
```

Valid Tcl: "Execute this intent procedure with this implementation"
Static property checks - Power domains code example

1. AON region must be designed and placed carefully to avoid DRC

   ![Diagram of AON region and power domains]

2. Code must be written with intent-implementation split

   ```python
   array set mflowgen.property.even_y {
   property "((aon_1y//y_pitch) % 2 == 0) & ((aon_ury//y_pitch) % 2 == 0)"
   describe "The AON region must begin and end on an even numbered standard cell row to prevent obstructed power switches and LUP DRCs"
   }
   array set mflowgen.property.max_width {
   property "((aon_ux - aon_llx) < horiz_switch_pitch)"
   describe "The always on region must be narrower than the pitch between power switch pitches so that no more than 1 column of switches is obstructed"
   }
   ```

3. Properties and static checks are re-used for AON region in a different design

   ![Diagram of re-used AON region in different design]
Other Static Check Constructs

- An **enum construct** checks annotations in the form
  - `mflowgen.enum.<setN:string>(<target:string>)`

- An **equality construct** has the form
  - `mflowgen.equality.<name:string>(<target:string>)`

For example, check that standard cell "INV_X1" used in a node (from a previous project) is a valid cell in the *current* technology.

For example, check in a 2D tile array that each tile (potentially heterogeneous) have same height
Complete Tool Flow

- **Common Modular Nodes**
- **Custom Modular Nodes**
- **Technology Interface Node**

**User DSL Code**

**Modular Flow Generator**

- **Python Graph-Building DSL Elaboration**

**Elaborated Graph** (Representing Assembled Physical Design Flow)

**Flow Consistency and Instrumentation Layer**
Christopher Torng - Composing Modular Flow Generators with Python-Based Static Checking to Enable Agile Principles in Physical Design

Complete Tool Flow

Common Modular Nodes

User DSL Code

Modular Flow Generator

Custom Modular Nodes

Python Graph-Building DSL Elaboration

Elaborated Graph
Representing Assembled Physical Design Flow

Technology Interface Node

Codebase Extraction

Tcl + Source Files

Flow Consistency and Instrumentation Layer
Complete Tool Flow

Common Modular Nodes → User DSL Code → Modular Flow Generator → Elaborated Graph

Technology Interface Node

Technology Consistency Static Checks
Design Consistency Static Checks
Property Checks (intent-implent split)

Codebase Extraction
Tcl + Source Files

Reports - Pass/Fail
Flow Consistency and Instrumentation Layer
Complete Tool Flow

User DSL Code

- Python Graph-Building DSL Elaboration

Modular Flow Generator

- Build System Code Generator (e.g., make)

User-Facing Physical Design Flow and supporting build system

Elaborated Graph Representing Assembled Physical Design Flow

- Codebase Extraction
  - Tcl + Source Files

Elaborated and Instrumented Graph Supports Assertions and Stashing Pre-Built Nodes

- Assertion Instrumentation (from node configuration spec)
  - Stash Instrumentation (disable edges for pre-built nodes)

Flow Consistency and Instrumentation Layer

- Reports - Pass/Fail

Technology Interface Node

Common Modular Nodes

- Technology Consistency Static Checks
- Design Consistency Static Checks
- Property Checks (intent-implement split)

(all nodes pass consistency checks)

Status:

- done -> 0 : constraints
- done -> 1 : freepdk-45nm
- build -> 2 : info
- build -> 3 : rtl
- build -> 4 : testbench
- build -> 5 : synopsys-dc-synthesis
- build -> 6 : cadence-innovus-flowsetup
- build -> 7 : verif_post_synth
- build -> 8 : cadence-innovus-init
- build -> 9 : cadence-innovus-power
- build -> 10 : cadence-innovus-place
- build -> 11 : cadence-innovus-cts
- build -> 12 : cadence-innovus-postcts_hold
- build -> 13 : cadence-innovus-route
- build -> 14 : cadence-innovus-postroute
- build -> 15 : cadence-innovus-postroute Hold
- build -> 16 : cadence-innovus-signoff
- build -> 17 : mentor-calibre-gdsmerge
- build -> 18 : synopsys-tpx-genlibdb
- build -> 19 : synopsys-vcs-sim
- build -> 20 : verif_post_layout
- build -> 21 : mentor-calibre-drc
- build -> 22 : mentor-calibre-lvs
- build -> 23 : synopsys-tp-power
Instrumenting Library Nodes with Assertions

Built-in Node Assertions

```yaml
name: synthesis
inputs: (...)
outputs: (...)
commands: (...)
preconditions:
  - assert
    File( 'inputs/design.v' )
postconditions
  - assert 'property'
    not in File( 'run.log' )
```

Assertions are Python snippets
Instrumenting Library Nodes with Assertions

**Built-in Node Assertions**

- `name: synthesis`
- `inputs: (...)`
- `outputs: (...)`
- `commands: (...)`

**Preconditions:**
- `assert`
  - `File('inputs/design.v')`

**Postconditions:**
- `assert 'property'`
  - `not in File('run.log')`

**Graph Visualization**

- `rtl`
- `tech`
- `design.v`
- `synthesis`
- `design.v`
- `tech`

- **Pytest fragments for preconditions**
- **Pytest fragments for postconditions**

- **Asserts are Python snippets**
- **FCI layer wraps each modular node**
Instrumenting Library Nodes with Assertions

Built-in Node Assertions

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>- assert File(‘inputs/design.v’)</td>
<td>- assert ‘property’ not in File(‘run.log’)</td>
</tr>
</tbody>
</table>

Design-Specific Assertions

- `syn = Step(...)`
- `syn.extend_preconditions([‘assert ...’])`
- `syn.extend_postconditions([‘assert ...’])`

Other modular nodes can be instrumented in a similar way.

Tweak dynamic assertions for different designs (e.g., assert clock-gating percent > 80%)

Assertions are Python snippets

FCI layer wraps each modular node
Instrumenting Graphs to Share Pre-Built Collateral
Instrumenting Graphs to Share Pre-Built Collateral

- ae39f5 [ May 20 ] authorA -- synth
- 5efe9c [ May 18 ] authorB -- rtl
- 3bfe2a [ May 17 ] authorA -- synth
- dace75 [ May 16 ] authorA -- synth
- be459c [ May 14 ] authorB -- rtl
- e4b5ac [ May 13 ] authorC -- floorplan

Front-end teammate pushes a built synthesis
Instrumenting Graphs to Share Pre-Built Collateral

The FCl layer modifies the graph and removes edges (it is now a vendor package)
We have built many chips that vary in complexity.

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>DenseAccel16</th>
<th>MiniCGRA</th>
<th>CryptoAccel</th>
<th>DenseAccelRRAM</th>
<th>RVMulticore</th>
<th>BaseSynch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Processing / ML</td>
<td>Image Processing</td>
<td>Cryptography</td>
<td>Machine Learning</td>
<td>General Purpose</td>
<td>Wireless</td>
<td></td>
</tr>
<tr>
<td>Technology</td>
<td>TSMC16</td>
<td>SKY130</td>
<td>SKY130</td>
<td>TSMC40</td>
<td>TSMC28</td>
<td>IBM180</td>
</tr>
<tr>
<td>Area</td>
<td>25mm²</td>
<td>10mm²</td>
<td>10mm²</td>
<td>29.2mm²</td>
<td>1.25mm²</td>
<td>2.31mm²</td>
</tr>
<tr>
<td>Max Frequency</td>
<td>750 MHz</td>
<td>60 MHz</td>
<td>325 MHz</td>
<td>200 MHz</td>
<td>500 MHz</td>
<td>20 MHz</td>
</tr>
<tr>
<td>Voltage</td>
<td>0.9 V</td>
<td>1.8 V</td>
<td>1.8 V</td>
<td>1.1 V</td>
<td>0.9 V</td>
<td>1.8 V</td>
</tr>
<tr>
<td>Power</td>
<td>0.5-1 W</td>
<td>-</td>
<td>-</td>
<td>126 mW</td>
<td>10s of mW</td>
<td>-</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>384 PE, 128 MEM</td>
<td>24 PE, 8 MEM</td>
<td>1</td>
<td>256 PE</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>On-Chip Memory</td>
<td>4.6 MB</td>
<td>8 KB</td>
<td>None</td>
<td>2MB/0.5MB RRAM/ SRAM</td>
<td>64 KB</td>
<td>None</td>
</tr>
<tr>
<td>Memory Levels</td>
<td>3</td>
<td>2</td>
<td>None</td>
<td>3</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>Has Off-Chip Memory</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Physical Hierarchies</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multiple Power Domains</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Number of Clock Domains</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from common library</td>
<td>30%</td>
<td>58%</td>
<td>94%</td>
<td>over 80%</td>
<td>86%</td>
<td>84%</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from previous designs</td>
<td>50%</td>
<td>24%</td>
<td>First design</td>
<td>First design</td>
<td>First design</td>
<td>First design</td>
</tr>
<tr>
<td>Months to tapeout</td>
<td>6</td>
<td>2.5</td>
<td>2.5</td>
<td>6</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>Static check runtime</td>
<td>2.2 sec</td>
<td>0.8 sec</td>
<td>0.2 sec</td>
<td>0.6 sec</td>
<td>&lt;1 sec</td>
<td>&lt;1 sec</td>
</tr>
</tbody>
</table>
We have built many chips that vary in complexity.

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>DenseAccel16</th>
<th>MiniCGRA</th>
<th>CryptoAccel</th>
<th>DenseAccelRRAM</th>
<th>RVMulticore</th>
<th>BaseSynch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Processing / ML</td>
<td>Image Processing</td>
<td>Cryptography</td>
<td>Machine Learning</td>
<td>General Purpose</td>
<td>Wireless</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technology</th>
<th>TSMC16</th>
<th>SKY130</th>
<th>SKY130</th>
<th>TSMC40</th>
<th>TSMC28</th>
<th>IBM180</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area</td>
<td>25mm²</td>
<td>10mm²</td>
<td>10mm²</td>
<td>29.2mm²</td>
<td>1.25mm²</td>
<td>2.31mm²</td>
</tr>
<tr>
<td>Max Frequency</td>
<td>750 MHz</td>
<td>60 MHz</td>
<td>325 MHz</td>
<td>200 MHz</td>
<td>500 MHz</td>
<td>20 MHz</td>
</tr>
<tr>
<td>Voltage</td>
<td>0.9 V</td>
<td>1.8 V</td>
<td>1.8 V</td>
<td>1.1 V</td>
<td>0.9 V</td>
<td>1.8 V</td>
</tr>
<tr>
<td>Power</td>
<td>0.5-1 W</td>
<td>-</td>
<td>-</td>
<td>126 mW</td>
<td>10s of mW</td>
<td>-</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>384 PE, 128 MEM</td>
<td>24 PE, 8 MEM</td>
<td></td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>On-Chip Memory</td>
<td>4.6 MB</td>
<td>8 KB</td>
<td>112 MB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory Levels</td>
<td>3</td>
<td>2</td>
<td>N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Has Off-Chip Memory</td>
<td>Yes</td>
<td>No</td>
<td></td>
<td></td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Physical Hierarchies</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multiple Power Domains</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Number of Clock Domains</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td></td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from common library</td>
<td>30%</td>
<td>58%</td>
<td>94%</td>
<td>over 80%</td>
<td>86%</td>
<td>84%</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from previous designs</td>
<td>50%</td>
<td>24%</td>
<td>First design</td>
<td>First design</td>
<td>First design</td>
<td>First design</td>
</tr>
<tr>
<td>Months to tapeout</td>
<td>6</td>
<td>2.5</td>
<td>2.5</td>
<td>6</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>Static check runtime</td>
<td>2.2 sec</td>
<td>0.8 sec</td>
<td>0.2 sec</td>
<td>0.6 sec</td>
<td>&lt;1 sec</td>
<td>&lt;1 sec</td>
</tr>
</tbody>
</table>

Different application domains

Technology from 16nm to 180nm
We have built many chips that vary in complexity. Here is a comparison of different chips and their specifications:

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>DenseAccel16</th>
<th>MiniCGRA</th>
<th>CryptoAccel</th>
<th>DenseAccelRRAM</th>
<th>RVMulticore</th>
<th>BaseSynch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image Processing / ML</td>
<td>TSMC16</td>
<td>SKY130</td>
<td>SKY130</td>
<td>TSMC40</td>
<td>TSMC28</td>
<td>IBM180</td>
</tr>
<tr>
<td>Complexity (macros)</td>
<td>25mm²</td>
<td>10mm²</td>
<td>10mm²</td>
<td>29.2mm²</td>
<td>1.25mm²</td>
<td>2.31mm²</td>
</tr>
<tr>
<td>Physical Design</td>
<td>750 MHz</td>
<td>60 MHz</td>
<td>325 MHz</td>
<td>200 MHz</td>
<td>500 MHz</td>
<td>20 MHz</td>
</tr>
<tr>
<td>Power</td>
<td>0.9 V</td>
<td>1.8 V</td>
<td>1.8 V</td>
<td>1.1 V</td>
<td>0.9 V</td>
<td>1.8 V</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>0.5-1 W</td>
<td>-</td>
<td>-</td>
<td>126 mW</td>
<td>10s of mW</td>
<td>-</td>
</tr>
<tr>
<td>On-Chip Memory</td>
<td>384 PE, 128 MEM</td>
<td>24 PE, 8 MEM</td>
<td>None</td>
<td>2MB/0.5MB RRAM/SRAM</td>
<td>64 KB</td>
<td>None</td>
</tr>
<tr>
<td>Memory Levels</td>
<td>4.6 MB</td>
<td>8 KB</td>
<td>None</td>
<td>None</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Has Off-Chip Memory</td>
<td>3</td>
<td>2</td>
<td>None</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Physical Hierarchies</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Multiple Power Domains</td>
<td>Yes</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Number of Clock Domains</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from common library</td>
<td>94%</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Months to tapeout</td>
<td>6</td>
<td>2.5</td>
<td>2.5</td>
<td>6</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>Static check runtime</td>
<td>2.2 sec</td>
<td>0.8 sec</td>
<td>0.2 sec</td>
<td>0.6 sec</td>
<td>&lt;1 sec</td>
<td>&lt;1 sec</td>
</tr>
</tbody>
</table>

- **Physical Design Complexity (macros)**
- **Multiple power domains and clock domains**
- **Both flat / hierarchical design flows**
We have built many chips that vary in complexity.

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>DenseAccel16</th>
<th>MiniCGRA</th>
<th>CryptoAccel</th>
<th>DenseAccelRRAM</th>
<th>RVMulticore</th>
<th>BaseSynch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>TSMC16</td>
<td>SKY130</td>
<td>SKY130</td>
<td>TSMC40</td>
<td>TSMC28</td>
<td>IBM180</td>
</tr>
<tr>
<td>Area</td>
<td>25mm²</td>
<td>10mm²</td>
<td>10mm²</td>
<td>29.2mm²</td>
<td>1.25mm²</td>
<td>2.31mm²</td>
</tr>
<tr>
<td>Max Frequency</td>
<td>325 MHz</td>
<td>200 MHz</td>
<td>500 MHz</td>
<td>1.25 V</td>
<td>0.9 V</td>
<td>1.8 V</td>
</tr>
<tr>
<td>Voltage</td>
<td>1.8 V</td>
<td>1.1 V</td>
<td>0.9 V</td>
<td>1.8 V</td>
<td>1.8 V</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td>-</td>
<td>126 mW</td>
<td>10s of mW</td>
<td>-</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>Number of PE</td>
<td>1</td>
<td>256 PE</td>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>On-Chip Memory</td>
<td>None</td>
<td>2MB/0.5MB RRAM/SRAM</td>
<td>64 KB</td>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td>None</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Has Offchip Memory</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Physical Hierarchies</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multiple Power Domains</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Number of Clock Domains</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from common library</td>
<td>30%</td>
<td>58%</td>
<td>94%</td>
<td>over 80%</td>
<td>86%</td>
<td>84%</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from previous designs</td>
<td>50%</td>
<td>24%</td>
<td>First design</td>
<td>First design</td>
<td>First design</td>
<td>First design</td>
</tr>
<tr>
<td>Months to tapeout</td>
<td>2</td>
<td>1.5</td>
<td>&lt;1 sec</td>
<td>&lt;1 sec</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static check runtime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Code Reuse of Common Library**

This reuse metric is high, but less challenging. It is straightforward to reuse generic code (e.g., Synopsys / Cadence / Hammer flows).

**Code Reuse in Second+ Designs**

This reuse metric is more challenging to make high.
We have built many chips that vary in complexity: AHA SoC, Stanford EE272, Raina Group SoC, Two Cornell Chips.

<table>
<thead>
<tr>
<th>Application Domain</th>
<th>DenseAccel16</th>
<th>MiniCGRA</th>
<th>CryptoAccel</th>
<th>DenseAccelRRAM</th>
<th>RVMultiCore</th>
<th>BaseSynch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Technology</td>
<td>TSMC16</td>
<td>SKY130</td>
<td>SKY130</td>
<td>TSMC40</td>
<td>TSMC28</td>
<td>IBM180</td>
</tr>
<tr>
<td>Area</td>
<td>25mm²</td>
<td>10mm²</td>
<td>10mm²</td>
<td>29.2mm²</td>
<td>1.25mm²</td>
<td>2.31mm²</td>
</tr>
<tr>
<td>Max Frequency</td>
<td>750 MHz</td>
<td>60 MHz</td>
<td>325 MHz</td>
<td>200 MHz</td>
<td>500 MHz</td>
<td>20 MHz</td>
</tr>
<tr>
<td>Voltage</td>
<td>0.9 V</td>
<td>1.8 V</td>
<td>1.8 V</td>
<td>1.1 V</td>
<td>0.9 V</td>
<td>1.8 V</td>
</tr>
<tr>
<td>Power</td>
<td>0.5-1 W</td>
<td>-</td>
<td>-</td>
<td>126 mW</td>
<td>10s of mW</td>
<td>-</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>384 PE, 128 MEM</td>
<td>24 PE, 8 MEM</td>
<td>1</td>
<td>256 PE</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>On-Chip Memory</td>
<td>4.6 MB</td>
<td>8 KB</td>
<td>None</td>
<td>2MB/0.5MB RRAM/SRAM</td>
<td>64 KB</td>
<td>None</td>
</tr>
<tr>
<td>Memory Levels</td>
<td>3</td>
<td>2</td>
<td>None</td>
<td>3</td>
<td>1</td>
<td>None</td>
</tr>
<tr>
<td>Has Off-Chip Memory</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Physical Hierarchies</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Multiple Power Domains</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Number of Clock Domains</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from common library</td>
<td>30%</td>
<td>58%</td>
<td>94%</td>
<td>over 80%</td>
<td>86%</td>
<td>84%</td>
</tr>
<tr>
<td>% of Codebase (LoC) reused from previous designs</td>
<td>50%</td>
<td>24%</td>
<td>First design</td>
<td>First design</td>
<td>First design</td>
<td>First design</td>
</tr>
<tr>
<td>Months to tapeout</td>
<td>6</td>
<td>2.5</td>
<td>2.5</td>
<td>6</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>Static check runtime</td>
<td>2.2 sec</td>
<td>0.8 sec</td>
<td>0.2 sec</td>
<td>0.6 sec</td>
<td>&lt;1 sec</td>
<td>&lt;1 sec</td>
</tr>
</tbody>
</table>

Base time to inspect entire PD codebase for static checks. Must add evaluation time of ~20 seconds per 1000 checks.
Code Reuse in Second+ Designs - MiniCGRA (SKY130)

- Overall code reuse remained high
- Code that is modified, removed, and added could be done in a modular fashion (many nodes do not need to be "touched" at all)
- Code is reused from both the DenseAccel16 design and the common libs
- High-effort design-for-reuse made a significant difference
Discussion Points

- **Designing for reuse** -- is not easy and can produce friction in the field
  - However, there is an *outsized return on investment* in physical design due to very long spin times

- **Effort** -- provide different investments to design for reuse
  - High-effort reusable code -- robustly **execute** design intent in different scenarios
  - Low-effort static property checks -- simply **lock** the design intent in current scenario

- **Can expressions be evaluated statically?** -- Most of the time, the tool is not necessary to calculate values

- **Scope of static checks** -- We are checking a codebase scattered across tools, jumping in and out of memory

- **Soft benefits** -- Easier to modify a modular node than a tangled system, better integration from teaching to research grade flows, accessibility to new hardware audience

- **Physical design and PL** -- Opportunities for gradual typing, aspect-oriented programming, metaprogramming
Takeaway Points

**Key Question:** How do we reuse code from different projects, built in different technologies, designed with different vendors?

Composing modular flow generators with Python-based static checking...

- can allow us to **isolate reusable code**
- and provide **quick feedback on the consistency guarantee** of the composition
Backup Slides
Amber SoC (TSMC16) - Code Reuse

- Percentage of common library node that is reused
- Percentage of final node that is custom
How do you compare two designs predictably?

**Key Idea:** Compare at a point where the final optimization effort is similar (similar slope = pushed to a similar extent)