

Garnet

NEXT-GENERATION HARDWARE GENERATORS IN PYTHON

Raj Setaluri, Alex Carsello

Hardware design process should
capture more than just RTL

Need to address all 4 at design time

1. Logical Design
2. Physical Design
3. Testing/Verification
4. Software API

Garnet: Next generation generator

- Generators as software objects which expose relevant methods for each view
- Modular and reusable classes
- Staged generation
- Generator-level passes
- **Generator is single source of truth**

What we're using now: Garnet (consolidate w/ prev slide)

- Dynamic generator layer on top of Magma (Python-based HDL)
- Each generator is a python object
- Can perform multiple passes on generator objects
- Objects serve as single source of truth for collateral generation

```
import pe, memory

def create_column(height, ...):
    pes = [pe(...) for _ in range(height)]
    mems = [mem(...) for _ in range(height)]
    for i in range(height):
        wire(pes[i].east, mems[i].west)
        wire(pes[i].south, pes[i + 1].north)
    return column(pes, mems)

def cgra(col_height, num_cols, ...):
    cols = []
    for i in range(num_cols):
        cols.append(create_column(col_height, ...))
        wire(cols[i - 1], cols[i])

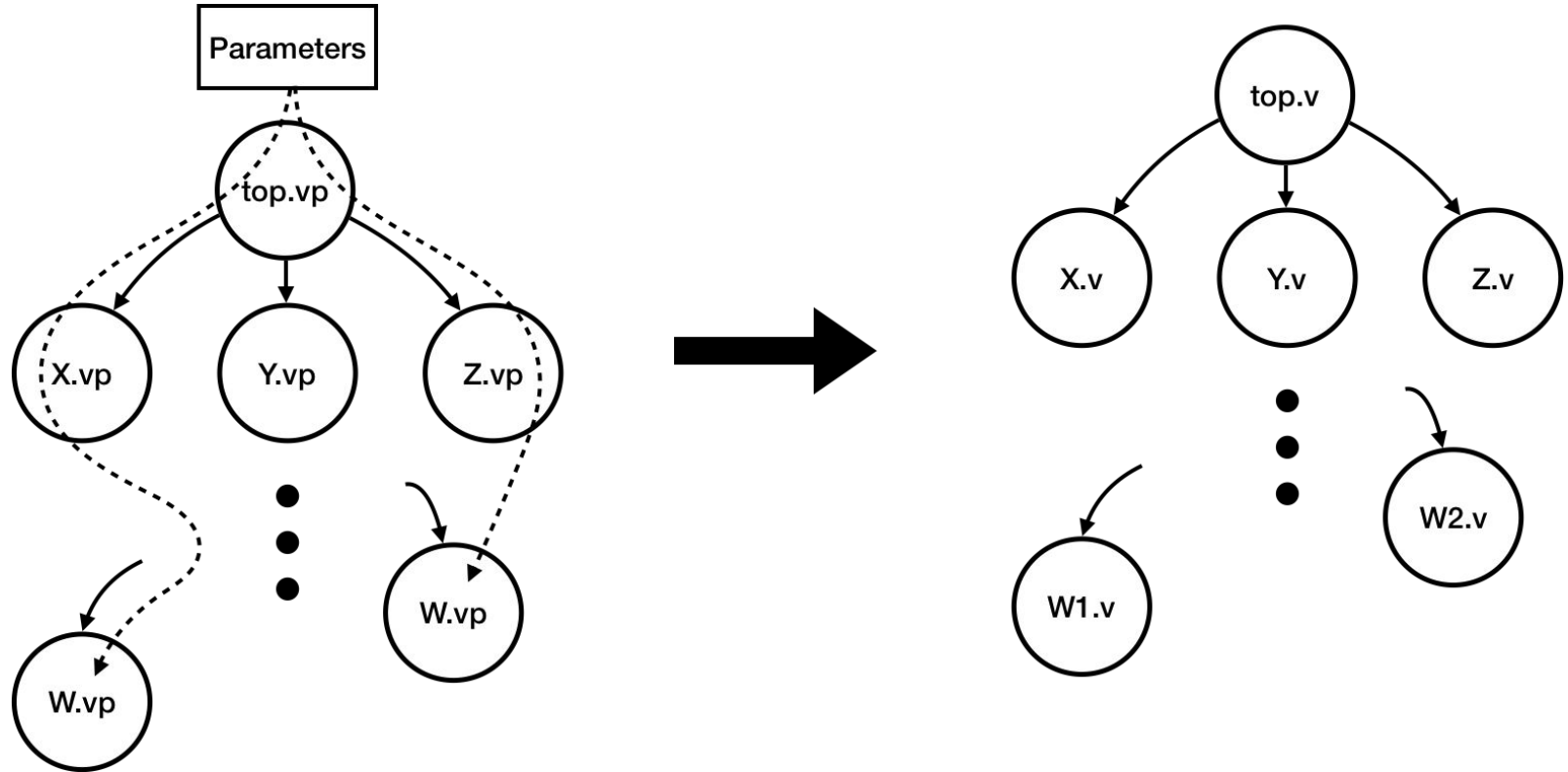
if __name__ == "__main__":
    col_height, num_cols, ... = get_opts()
    my_cgra = cgra(col_height, num_cols, ...)
    my_cgra.generate_verilog()
```

This work is in its very
early stages!

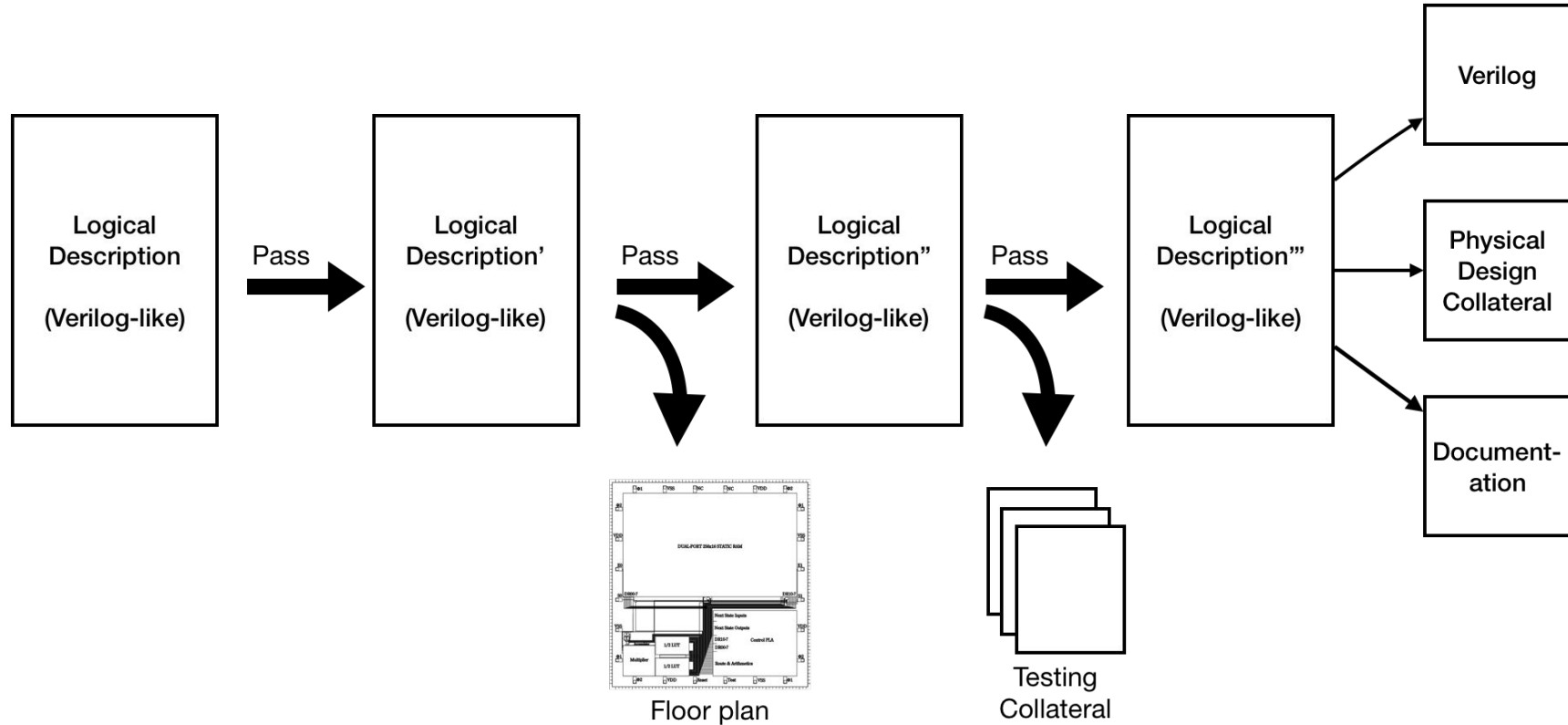
We'd appreciate any
feedback you have on
our approach.



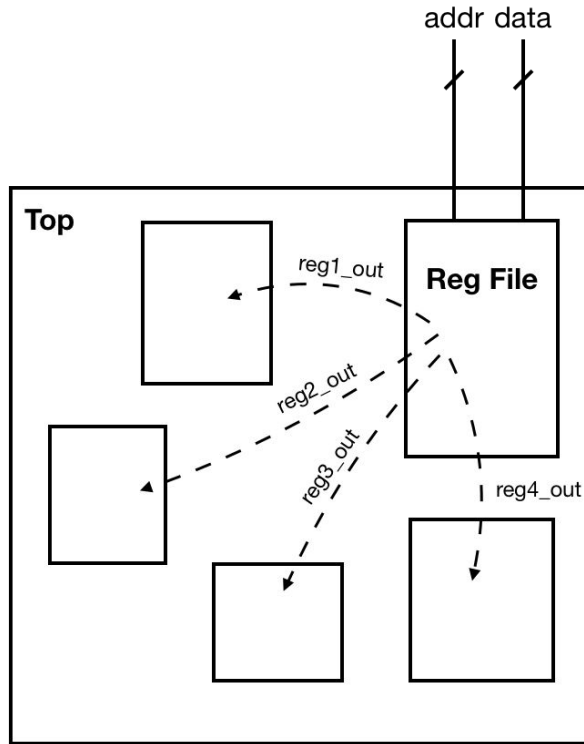
Single pass generator



Staged generator

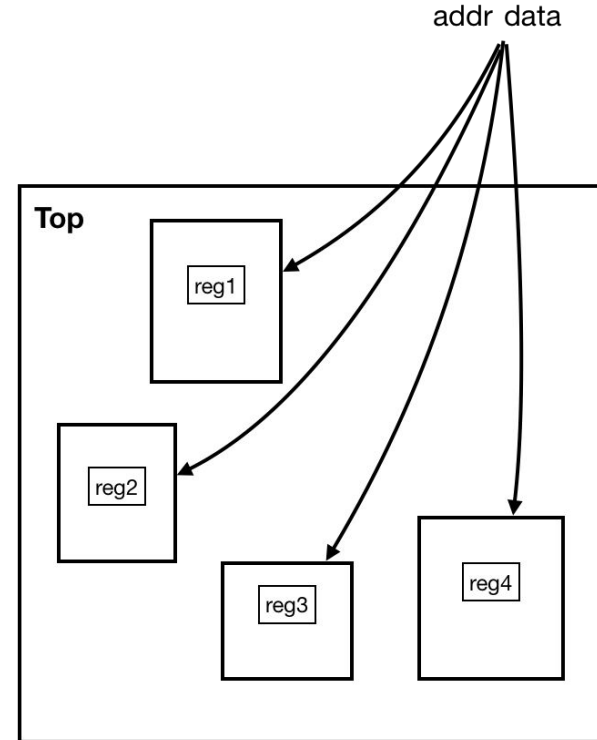


Staged generator - Register file example



Abstract Logical Description

Pass



Physical Implementation

- Something about why staged generation is cool and what it enables

Addressing Logical Design

- In generators like Genesis2, parameters are limited to simple types
- Garnet generators can take more complex parameters, e.g. another generator
- Re-use, abstraction, and parameters that make sense!
- For example, tile generator:

```
class Tile(generator.Generator):
    def __init__(self, core):
        super().__init__()

        self.core = core
        self.sb = SB(self.core.outputs())
        widths = [get_width(i.type()) for i in self.core.inputs()]
        self.cbs = [CB(10, w) for w in widths]
```

Case Study: Tile Generator

- Takes in arbitrary core generator as input
- Examines its interface to instantiate CB's and SB's properly
- Examines its features to generate hardware for reading and writing feature addresses properly
- In Genesis2, we had entirely separate tile generators for each different core

```
class Tile(generator.Generator):  
    def __init__(self, core):  
        super().__init__()  
  
        self.core = core  
        self.sb = SB(self.core.outputs())  
        widths = [get_width(i.type()) for i in self.core.inputs()]  
        self.cbs = [CB(10, w) for w in widths]
```

Addressing Physical Design Issues

- Key idea: **decouple the logical design from physical implementation**
 - Global signals are an example
- Generate efficient physical implementations while keeping logical intent clear and concise
- Prevent errors introduced when encoding physical implementation in RTL
- Abstractions for clock domains and power domains

Case Study: Global Signal Management

- Logical intent is simple: expose some net to everybody
 - E.g., top level configuration register address bus
- Physical implementation is much more complex
 - We have to insert structures to our design to distribute these signals in a physically feasible manner, e.g. trees, bristle
- Introduce a set of reusable directives for various strategies
 - Orthogonal to remainder of design

Case Study: Global Signal Management

```
class ColumnBase:
    def __init__(self, tiles, ...):
        self.tiles = tiles
        self.height = len(tiles)

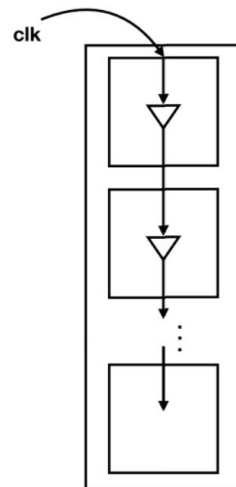
    # Basic logic for column module.
    # ...
```

```
    # Wire tiles together.
    for i in range(self.height - 1):
        self.wire(self.tiles[i].O, self.tiles[i + 1].I)
        self.wire(self.tiles[i].I, self.tiles[i + 1].O)
```

```
@abstractmethod
def wire_clocks(self):
    pass
```

```
class ColumnNaive(ColumnBase):
    def wire_clocks(self):
        for tile in self.tiles:
            self.wire(self.ports.clk, tile.ports.clk)
```

```
class ColumnMeso(ColumnBase):
    def wire_clocks(self):
        self.wire(self.ports.clk, self.tiles[0].ports.clk)
        for i, tile in self.tiles:
            if i < len(self.tiles):
                tile.add_port("clk_out", Clock)
                tile.wire(tile.ports.clk, tile.ports.clk_out)
                self.wire(tile.ports.clk_out, self.tiles[i + 1].ports.clk)
```



- Can use inheritance to decouple logical intent from physical implementation
- Create separate functions for issues related to physical implementation

Case Study: Global Signal Management

```
class Top:
    def __init__(self):
        # Some logic
        # ...

        config_manager = ConfigManager(...)

        make_global(config_manager.ports.addr, "config_addr")
        make_global(config_manager.ports.data, "config_data")
        config_manager.ports.addr.distribute()
        config_manager.ports.dat.distribute()

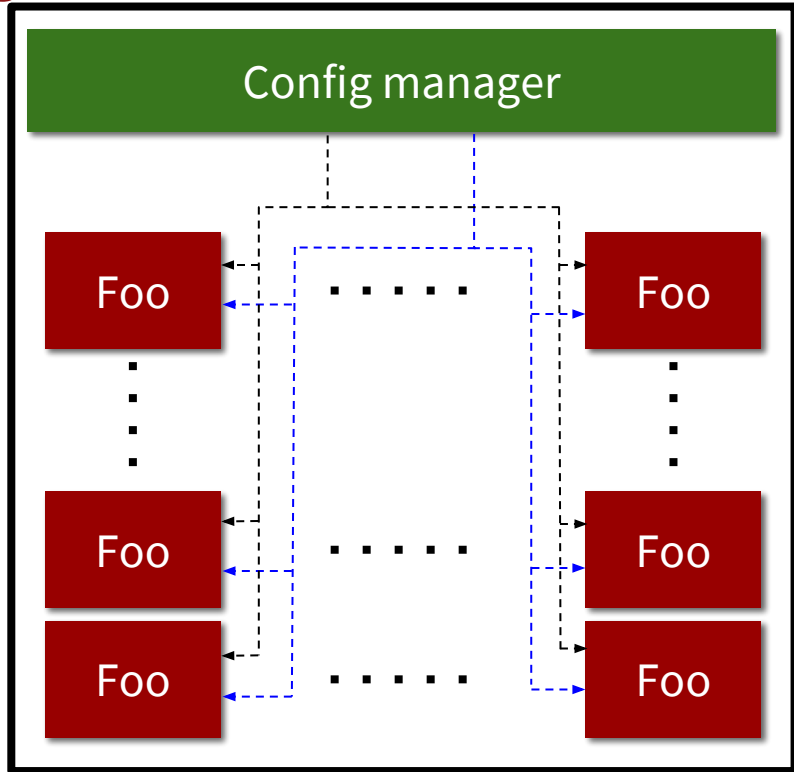
        foo = Foo(...)

        # ...

class Foo:
    def __init__(self):
        # ...

        config_reg = ConfigRegister(...)

        self.wire(globals_["config_addr"], config_reg.port.addr)
```



Addressing Physical Design

- Many parameter changes require changes to physical design scripts
- Generators should enable easy, **end-to-end** DSE
- Having to change PD scripts every time a parameter slows DSE cycles and limits re-usability
- Generator must extend param changes to all generated collateral, including PD scripts

Addressing Testing/Verification

- We need to start testing at the generator level, not just instances
- Generator outputs test collateral, affected by parameters and passes
- See upcoming talk for more...

Addressing SW API concern

- Logical design should be collocated with API level concerns
- System should allow HW designer to specify abstract components, for which both RTL **and** SW are generated
- Excel spreadsheets for config registers are an example of this, but...
 - Ideally it is collocated with the rest of the design
 - And natural to write inline

Case study: Configuration Management

- Expose high-level functions for read and write
 - Automates testing and provides API for bitstream generation
- Automatically generate global addresses
- Reconfiguration: one of the biggest sources of design complexity

```
class PE(generator):
    def __init__(self, ...):
        # ...

        # Add a read/write register of 5 bits named "op" at
        # addr 0x0.
        self.add_register("op", 5, "RW", 0x0)

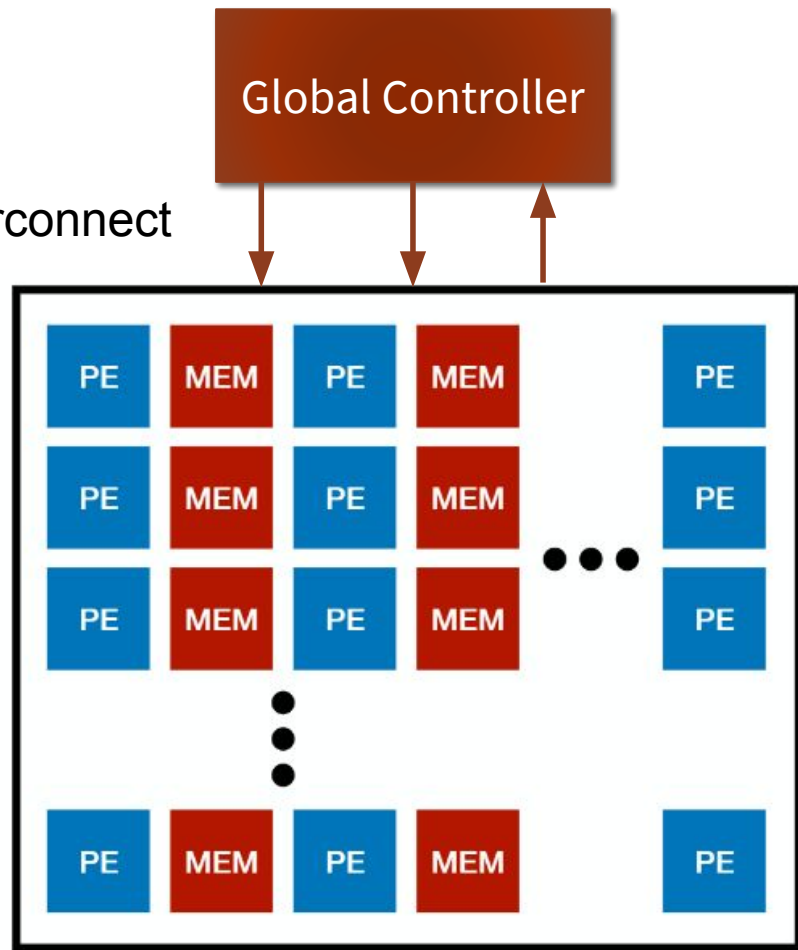
cgra = CGRA(16, 16)
cgra.tiles[0][0].core.write_op(0x3)
bitstream = cgra.generate_bitstream()
```

Conclusion

- Next-generation generators must support a more holistic approach to hardware design
 - Logical design
 - Physical design
 - Testing/Verification
 - Software API
- Allows **end-to-end** design space exploration
- Robust, flexible, trustable generators to facilitate extensive reuse

CGRA Architecture Overview

- Grid of tiles with reconfigurable interconnect
- Each tile has:
 - Core (memory or PE)
 - CBs: select input for core
 - SB: select tile outputs
 - Reconfiguration/debug stuff



CGRA Design Process

- Highly parameterized design using Genesis
- Taped out 16x16 design using TSMC 16nm
- Problems:
 - No parameter DSE because not understandable (logical)
 - No parameter DSE because breaks physical design (physical)
 - Realized global signals don't work for PD → had to rip up entire RTL
 - No reuse of reconfiguration logic

Problem 1

- No parameter DSE because not understandable (logical)
- Fragility (many parameter combinations just don't work)
- Params should be correct/valid by construction

Problem 2

- No parameter DSE because breaks physical design (physical)
- Don't want to reinvent the wheel for PD every time we change a parameter

Problem 3

- Realized global signals don't work for PD → had to rip up entire RTL

Problem 4

- No reuse of reconfiguration logic
- Instead, very similar logic implemented in slightly different ways
 - Address decode
 - Write enable
- Will address this with a configuration manager

What We Used Before: Genesis2

- Not a huge jump from writing Verilog
- Output RTL more readable/predictable
- Debugging easier (for now)
- Generator Readability
- Essentially just text manipulation. The generator doesn't have any information about what it's generating
- Easy to get mismatch between RTL output and collateral (violate single source of truth)

Genesis2 (Perl + Verilog)

```
//; for (my $h=0; $h<$cgra_grid_height; $h++) {  
//;   for (my $w=0; $w<$cgra_grid_width; $w++) {  
//;     my $tile_type = $tile_grid{$key};  
//;     if ($tile_type eq "mem") {  
//;       my $data_bus = $tile_config->  
//;         { $tile_type }{ 'gen mem for busname' };  
//;       my $bus_width = $bus_width_hash { $data_bus };  
//;       wire [`${bus_width}-1`:0] mem_chain`${h}``${w}`;  
//;       wire mem_chain_valid`${h}``${w}`;  
//;     }  
//;     if ((($tile_type eq "mem") || ($tile_type eq "pe")) {  
//;       for (my $i=0; $i<$global_signal_count; $i++) {  
//;         if (($w%2==0) && ($h%2==0)) {  
//;           wire global_wire_h2l_1`${i}``${w}``${h}`;  
//;         }  
//;       }  
//;       wire global_wire_l2h_0`${w}``${h}`;  
//;     }  
//;   }  
//; }
```

CGRA Top

```
class CGRA(generator.Generator):
    def __init__(self, width, height):
        super().__init__()

        self.global_controller = GlobalController(32, 32)
        columns = []
        for i in range(width):
            tiles = []
            for j in range(height):
                core = MemCore(16, 1024) if (i % 2) else PECore()
                tiles.append(Tile(core))
            columns.append(Column(tiles))
        self.interconnect = Interconnect(columns)
```


From interconnect:

```
for column in self.columns:
    self.wire(self.ports.config, column.ports.config)
    self.wire(self.ports.reset, column.ports.reset)
self.wire(self.ports.west, self.columns[0].ports.west)
self.wire(self.ports.east, self.columns[-1].ports.east)
for i, column in enumerate(self.columns):
    self.wire(self.ports.north[i], column.ports.north)
    self.wire(self.ports.south[i], column.ports.south)
for i in range(1, self.width):
    c0 = self.columns[i - 1]
    c1 = self.columns[i]
    for j in range(self.height):
        self.wire(c1.ports.west[j].0, c0.ports.east[j].I)
        self.wire(c0.ports.east[j].0, c1.ports.west[j].I)
```

In Garnet: top + column + interconnect = **177 lines of Python**

In Genesis2: CGRA top = **1071 lines of Verilog + Perl**

Example Clock Wiring Pass

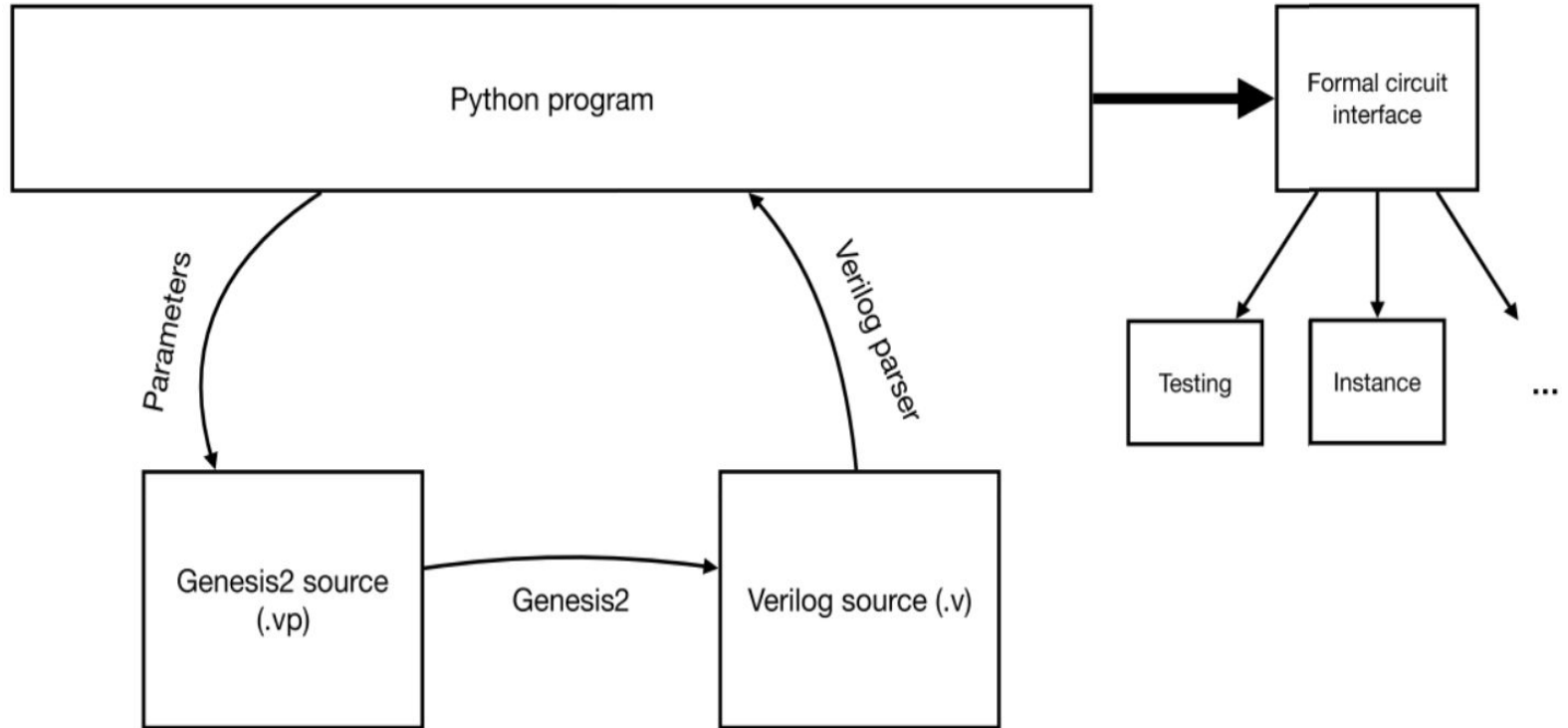
Pass 

```
def wire_clocks(generator):  
    if "clk" not in generator.ports:  
        generator.add_port("clk", In(Clock))  
    for child in generator.children():  
        wire_clocks(child)  
        generator.wire(generator.ports["clk"],  
                        child.ports["clk"])  
  
cgra = CGRA(16, 16)  
wire_clocks(cgra)  
  
cgra_circuit = cgra.generate_circuit()
```

Garnet CGRA

- Over the summer, we've been porting the Genesis2 CGRAGenerator to Garnet
- We currently have Garnet flow that outputs Verilog for full (slightly simplified) CGRA
- Key problems we're trying to address:
 - Low degree of re-use/abstraction
 - Not enough separation of concerns
 - Fragility
 - Poor test coverage
 - Unclear parameters

Wrapping Genesis2 Cores



Future Directions: Global Signal Management

```
def wire_cgra_clocks(cgra):
    cgra.add_port("clk", In(Clock))
    for column in cgra.columns:
        column.add_port("clk", In(Clock))
        cgra.wire(cgra.clk, column.clk)
        tiles = column.tiles

    # Naive strategy.
    for tile in tiles:
        wire_clocks(tile)
        column.wire(column.clk, tile.clk)

    # Smarter strategy.
    for i, tile in enumerate(tiles):
        wire_clocks(tile)

        if i == 0:
            column.wire(column.clk, tile.clk)
        else:
            column.wire(tiles[i - 1].clk_out, tile.clk)

    if i < len(tiles) - 1:
        tile.add_port("clk_out", Out(Clock))
        buff = Buffer()
        tile.wire(tile.clk, buff.I)
        tile.wire(buff.O, tile.clk_out)
```

