BFG

An Open-Source Silicon Compiler for Reconfigurable Fabrics

OSCAR Workshop 2026

Arya Reais-Parsi

Department of Electrical Engineering and Computer Science

University of California, Berkeley

Motivation

- **Accelerators are in vogue**
  - using FPGAs, right?
- **FPGAs enjoy niche use cases**
  - Emulation
  - Low-latency, high-throughput, low-volume
    - HFT, Automotive, Radio
- **FPGAs *promise* {fast, flexible, cheap}**
  - with *substantial* caveats

- **Untapped potential**
- **Unfulfilled promises**

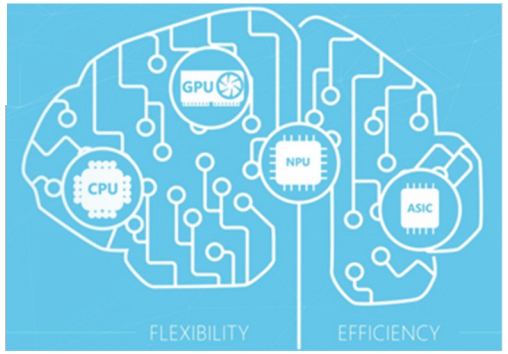
aws

Microsoft

Project Brainwave

FireSim

MiSTer FPGA



MAGMIO

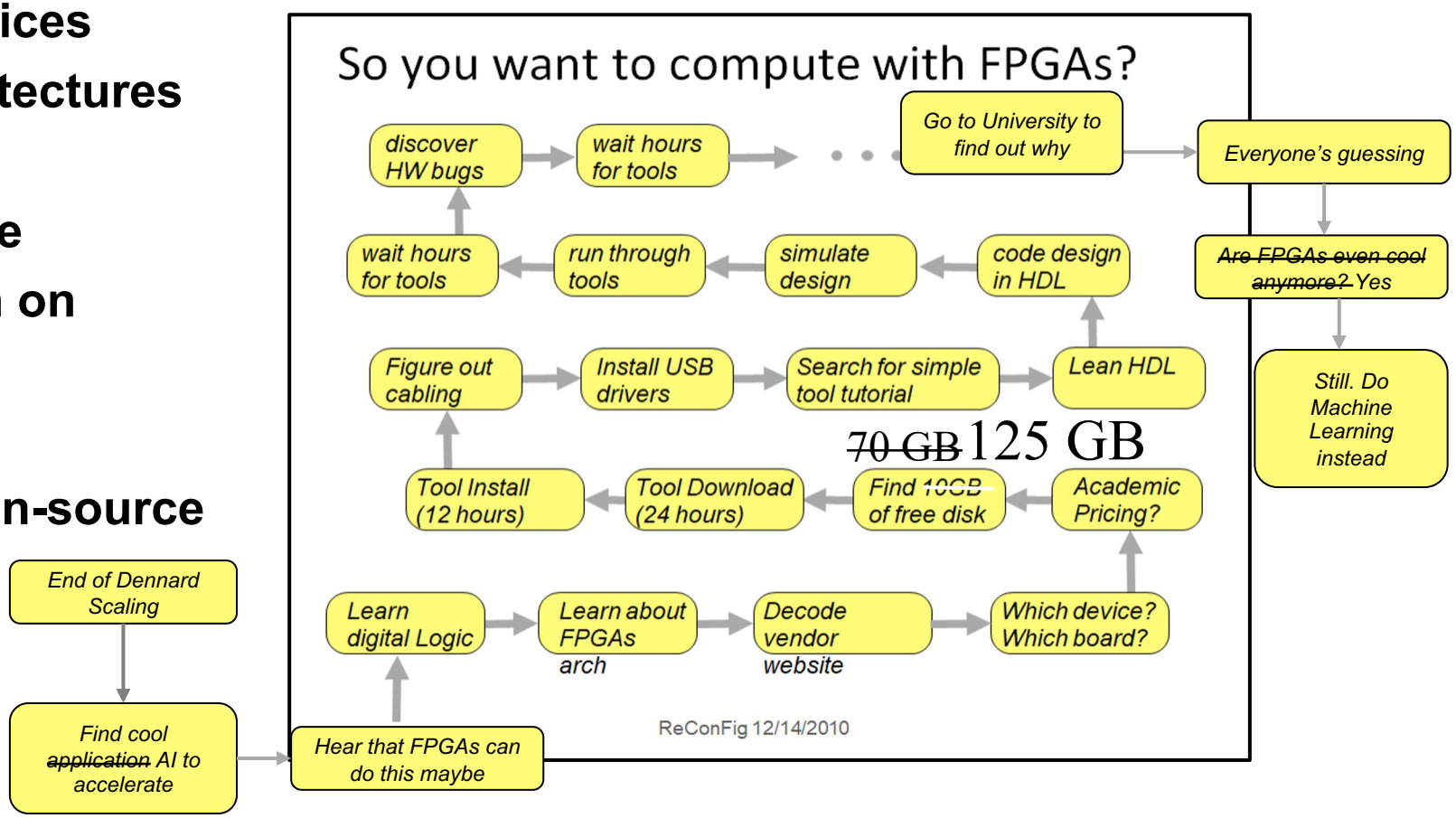
Jane Street

BAE SYSTEMS

FPGAs are still Too Hard

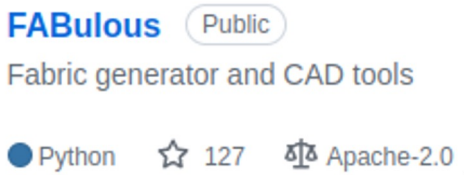
- **Myriad complicated devices**
- **Secret proprietary architectures**
- **Painful tooling**
- **Requires niche expertise**
- **Hard to do any research on**
- **Hard to innovate**

- **Want real, physical, open-source FPGA architectures**
  - Boon for architects, circuit designers, EDA, PL, security, …



eFPGA Generators

- **Others agree; have built eFPGA generators**
  - OpenFPGA, PRGA, FABulous



- **Rapid architectural exploration, real tape outs**
- **Generate RTL description of a fabric**
  - You implement in standard-cell flow
  - Process portable; easier to build; slower

But

- **FPGAs are not that complicated**
- **Can we do better with an open-source FPGA compiler?**

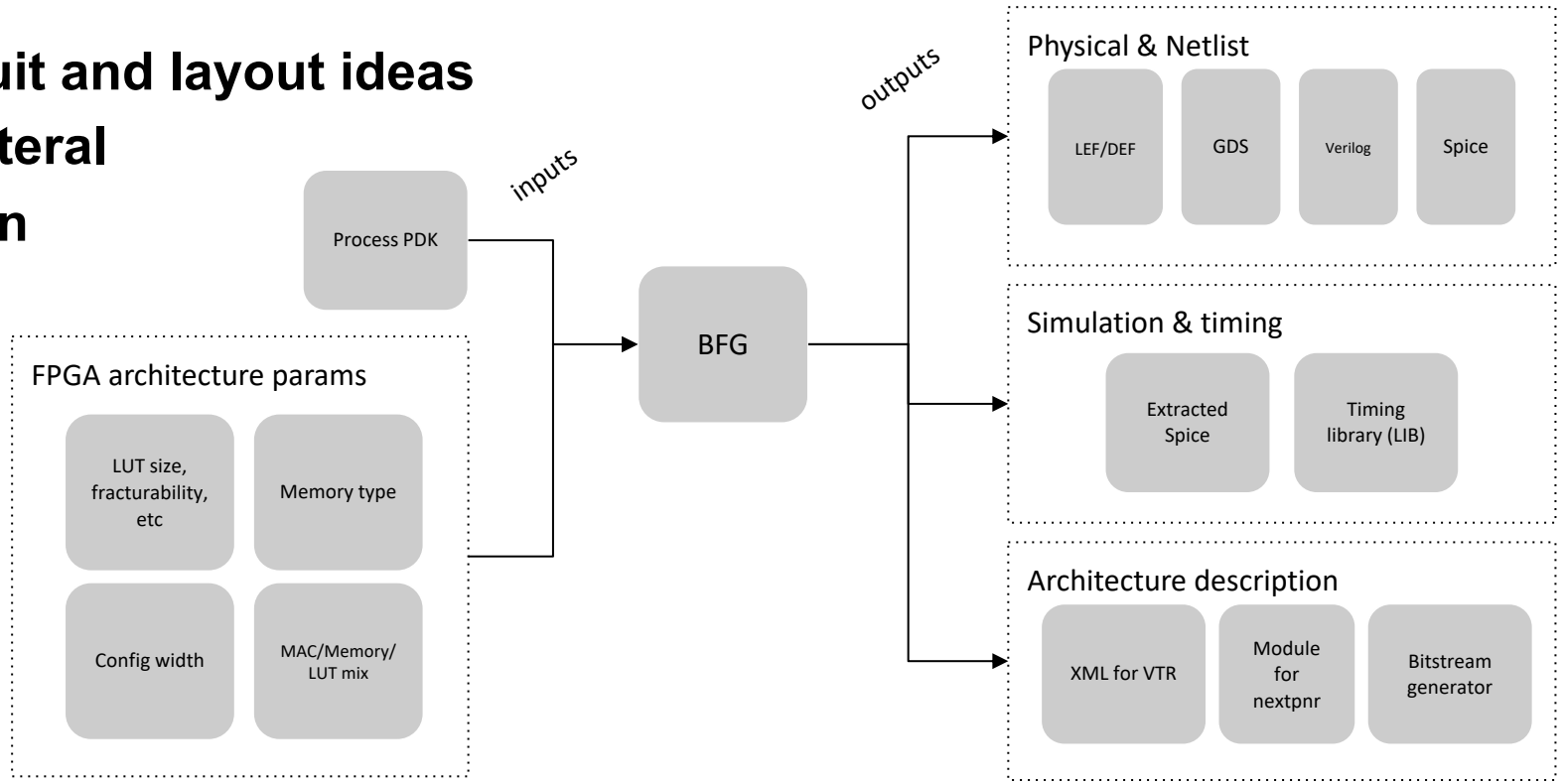
BFG

A parameterisable FPGA circuit + layout generator

- **produces performant IP and (eventually) whole fabrics**
- **no restrictive dependencies**
- **process portable***
- **a repository for AMS circuit and layout ideas**
- **outputs typical VLSI collateral**
- **DRC clean by construction**
- **LVS externally verified**

Real layout gives us real

- **parasitics**
- **timing, power**
- **logic density**



Contributions

1. **An open-source, C++ analog/mixed-signal (AMS) framework**
   - relying on vendor-agnostic middleware, VLSIR
   - geometry
   - circuit (netlist)
   - conveniences

2. **A starting collection of FPGA IP generators**
   - lower area, power, and delay than standard-cell versions

3. **An example of how to assemble a modern fabric**
   - based on literature, and
   - reverse-engineered UltraScale interconnect

BFG: C++ AMS Framework

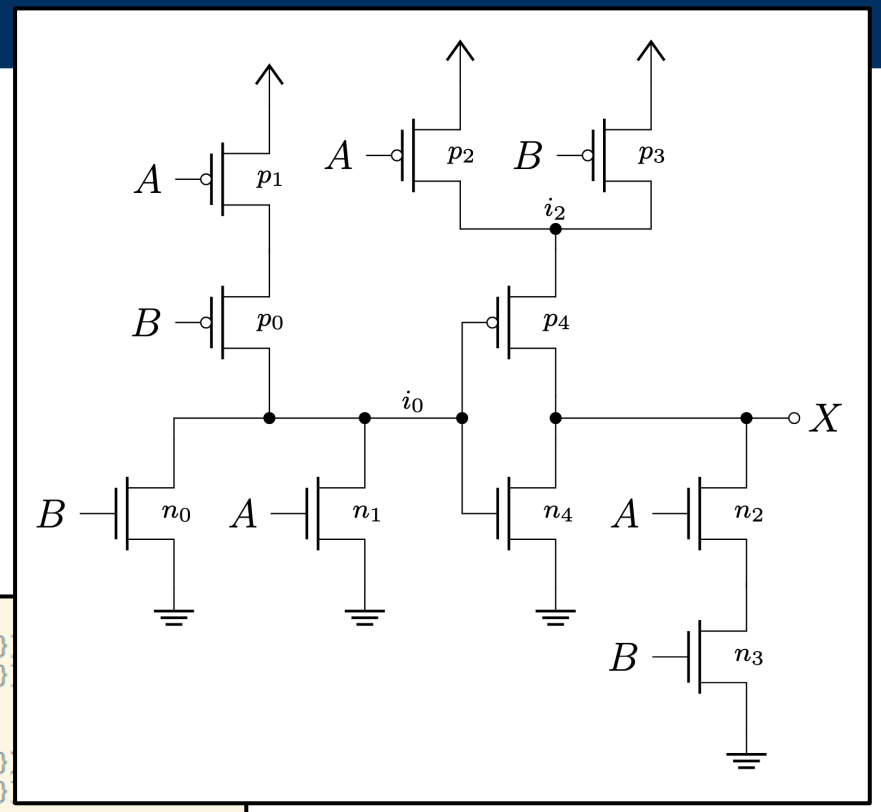
- **No better alternatives when we started**
  - script magic, klayout (free); script Virtuoso ($$$$)
  - OpenAccess is an oxymoron ($$)
- **C++ software library to assemble circuits and layout, with *conveniences***
  - A router; Layout guides
- **Hierarchically composes parameterised generators (“p-cells”)**
- **Vendor-agnostic middleware, VLSIR, for interchange formats**
  - Language-agnostic schemas (protobufs)
  - Read/write LEF/DEF, GDSII, Spice dialects, etc.
- **Now you have gdsfactory (Python) and Substrate (Rust) too**

Circuit (Netlist)

Assemble a circuit:

```
76  bfg::Circuit *nfet_01v8 =
77      design_db_->FindCellOrDie("sky130", "sky130_fd_pr__nfet_01v8")->circuit();
78
79  circuit::Instance *nfet_0 = circuit->AddInstance("nfet_0", nfet_01v8);
80  circuit::Instance *nfet_1 = circuit->AddInstance("nfet_1", nfet_01v8);
81  circuit::Instance *nfet_2 = circuit->AddInstance("nfet_2", nfet_01v8);
82  circuit::Instance *nfet_3 = circuit->AddInstance("nfet_3", nfet_01v8);
83  circuit::Instance *nfet_4 = circuit->AddInstance("nfet_4", nfet_01v8);
84
85  bfg::Circuit *pfet_01v8 =
86      design_db_->FindCellOrDie(
87          "sky130", "sky130_fd_pr__pfet_01v8_hvt")->circuit();
88  circuit::Instance *pfet_0 = circui
89  circuit::Instance *pfet_1 = circui
90  circuit::Instance *pfet_2 = circui
91  circuit::Instance *pfet_3 = circui
92  circuit::Instance *pfet_4 = circui
```

```
94   circuit::Wire i_0 = circuit->AddSignal("i_0");
95   nfet_0->Connect({{"d", i_0}, {"g", B}, {"s", VGND}, {"b", VNB}
96   nfet_1->Connect({{"d", i_0}, {"g", A}, {"s", VGND}, {"b", VNB}
97
98   circuit::Wire i_1 = circuit->AddSignal("i_1");
99   pfet_0->Connect({{"d", i_1}, {"g", B}, {"s", i_0}, {"b", VPB}}
100  pfet_1->Connect({{"d", VPWR}, {"g", A}, {"s", i_1}, {"b", VPB}
101
102  circuit::Wire i_2 = circuit->AddSignal("i_2");
103  pfet_2->Connect({{"d", VPWR}, {"g", A}, {"s", i_2}, {"b", VPB}});
104  pfet_3->Connect({{"d", VPWR}, {"g", B}, {"s", i_2}, {"b", VPB}});
105
106  nfet_4->Connect({{"d", X}, {"g", i_0}, {"s", VGND}, {"b", VNB}});
107  pfet_4->Connect({{"d", i_2}, {"g", i_0}, {"s", X}, {"b", VPB}});
108
109  circuit::Wire i_3 = circuit->AddSignal("i_3");
110  nfet_2->Connect({{"d", X}, {"g", A}, {"s", i_3}, {"b", VNB}});
111  nfet_3->Connect({{"d", i_3}, {"g", B}, {"s", VGND}, {"b", VNB}});
```



atoms/sky130_xor2.cc

Geometry

- **Manipulate layout:**
  - Add rectangles, (weird!) polygons on different layers
  - Lookup PDK rules in database
  - Compose sub-cells together
  - Translate, rotate, add margin, etc
  - Test distances, intersections, etc
  - Unit grid for alignment

- **Convenience features:**
  - PolyLines with bulges around vias turn into wires
  - Arrange sub-cells in Rows, Checkerboard, etc
  - Handle weird polygons (hard)
  - What else…?

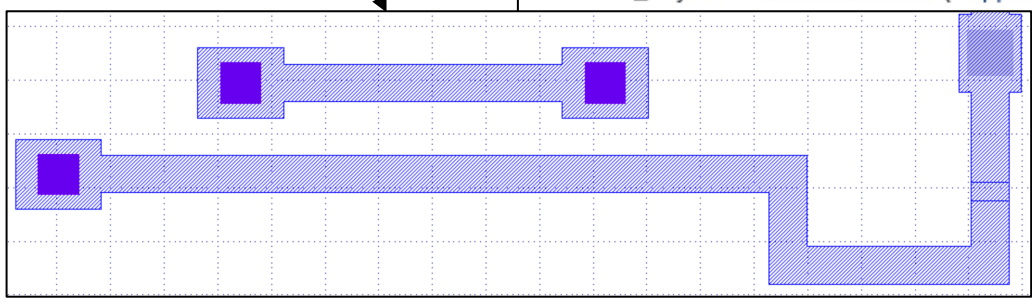
```
const auto &poly_rules = db.Rules("poly.drawing");
const auto &mcon_rules = db.Rules("mcon.drawing");
const auto &met1_rules = db.Rules("met1.drawing");
const auto &li_ncon_rules = db.Rules("li.drawing", "
const auto &li_pcon_rules = db.Rules("li.drawing", "
const auto &li_polycon_rules = db.Rules("li.drawing"
```

```
int64_t poly_contact_to_ndiff =
    ndiff_polycon_rules.min_separation + polycon_rules.via_width / 2;
int64_t poly_contact_to_pdiff =
    pdiff_polycon_rules.min_separation + polycon_rules.via_width / 2;

int64_t via_centre_to_poly_edge =
    polycon_rules.via_width / 2 + poly_polycon_rules.min_separation;

int64_t poly_overhang = li_rules.min_width + li_rules.min_separation;
int64_t poly_pitch = poly_rules.min_pitch;
int64_t poly_gap = std::max(
    poly_pitch - poly_rules.min_width,
    std::max(ncon_rules.via_width, pcon_rules.via_width) +
        2 * std::max(poly_ncon_rules.min_separation,
                     poly_pcon_rules.min_separation));

int64_t gap_top_y = std::min(
    main_layout->GetPointOrDie("upper_left.column_2_centre_bottom").y
    _right.column_2_centre_bottom").
    _left.column_2_centre_bottom").y
    _right.column_2_centre_bottom").
    y_rules.min_separation;
```

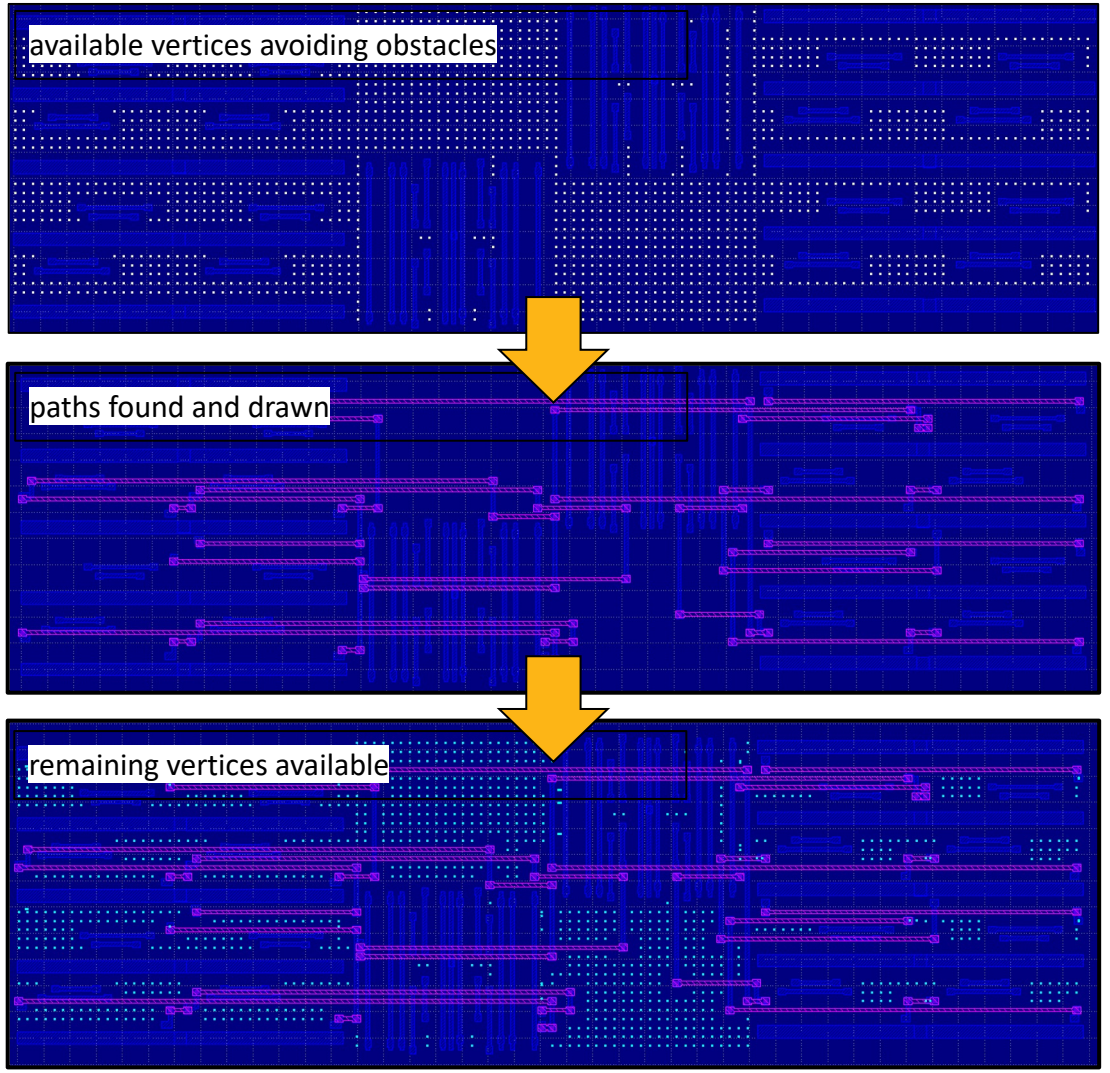


Routing

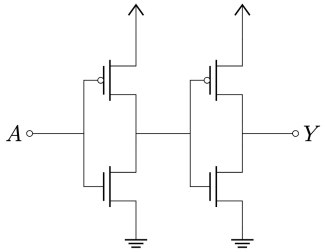
Minimum viable router

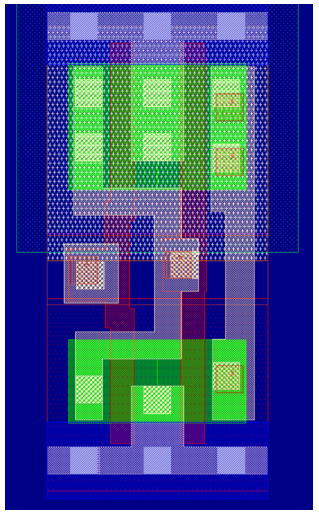
- Define a grid from PDK parameters
- Avoids obstacles
- Successively connect point to point
  - Or point to an existing net
- Order matters: layer optimisations on top of single route calls
- Grid alignment not required!
- Supports non-linear edge cost! $C(L) \propto L^2$
- What else…?

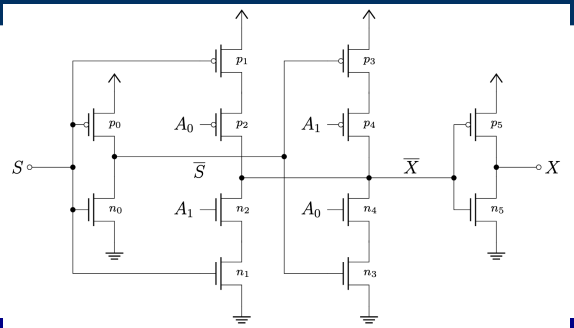
Available as a gRPC service

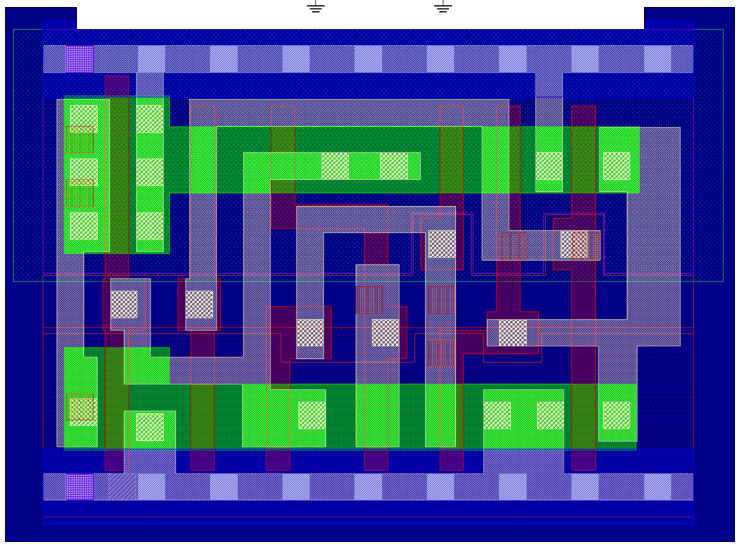


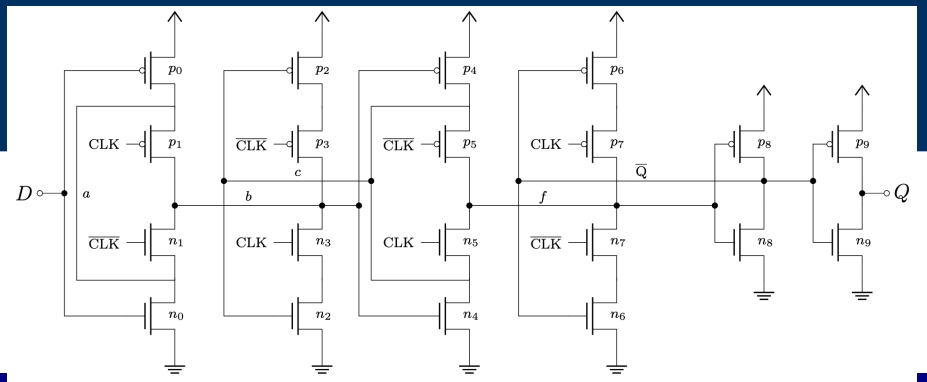
Parameterised FPGA IP (sky130)

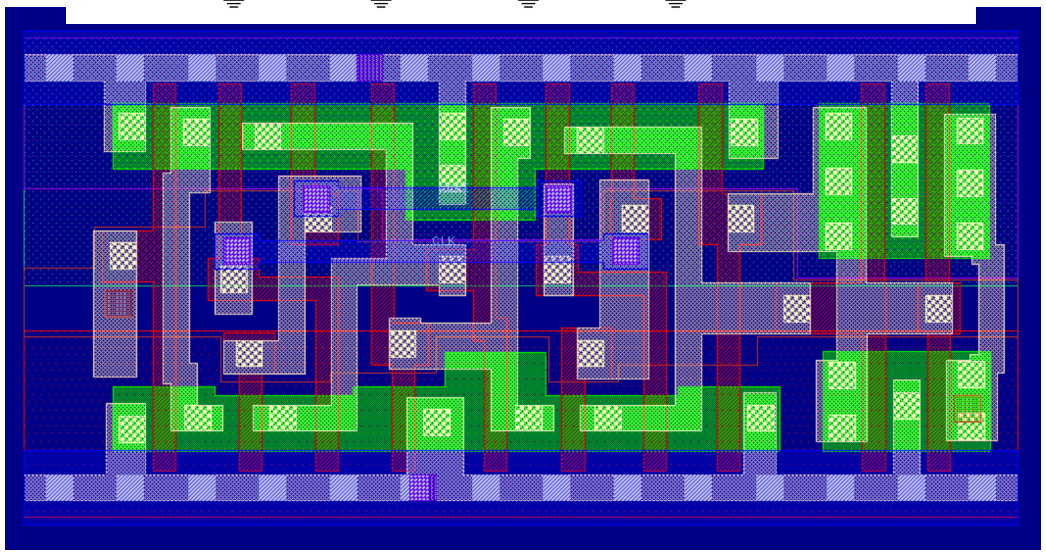












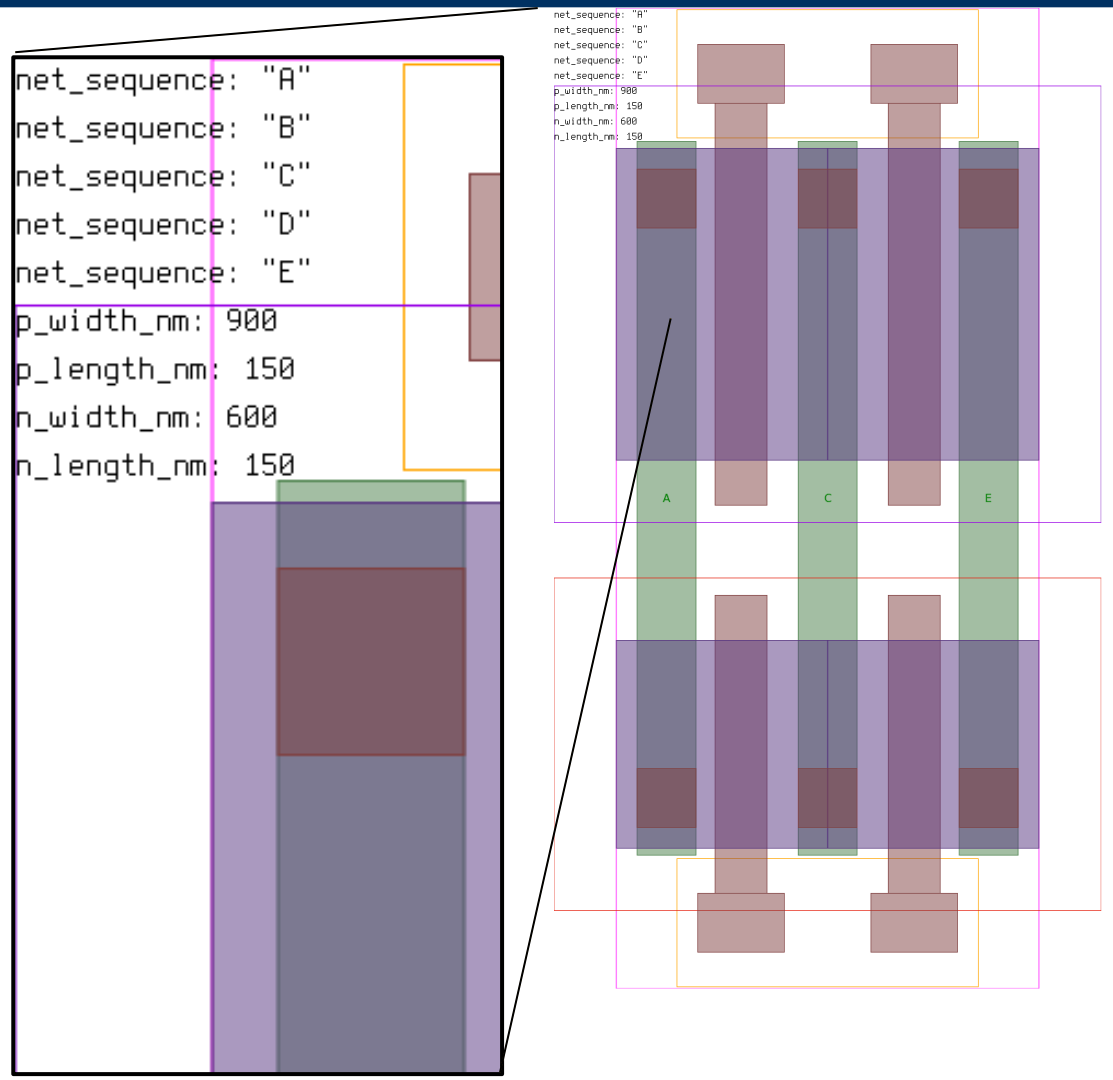
```
buf_params = {
    .height_nm = 2720,
    .width_nm = 460,
    .nfet_0_width_nm = 520,
    .nfet_1_width_nm = 520,
    .pfet_0_width_nm = 790,
    .pfet_1_width_nm = 790,
    ...
};
```

```
mux2_params = {
    .height_nm = 2720,
    .width_nm = 460,
    .nfet_0_width_nm = 420,
    .nfet_1_width_nm = 360,
    ...
    .pfet_0_width_nm = 420,
    .pfet_1_width_nm = 420,
```

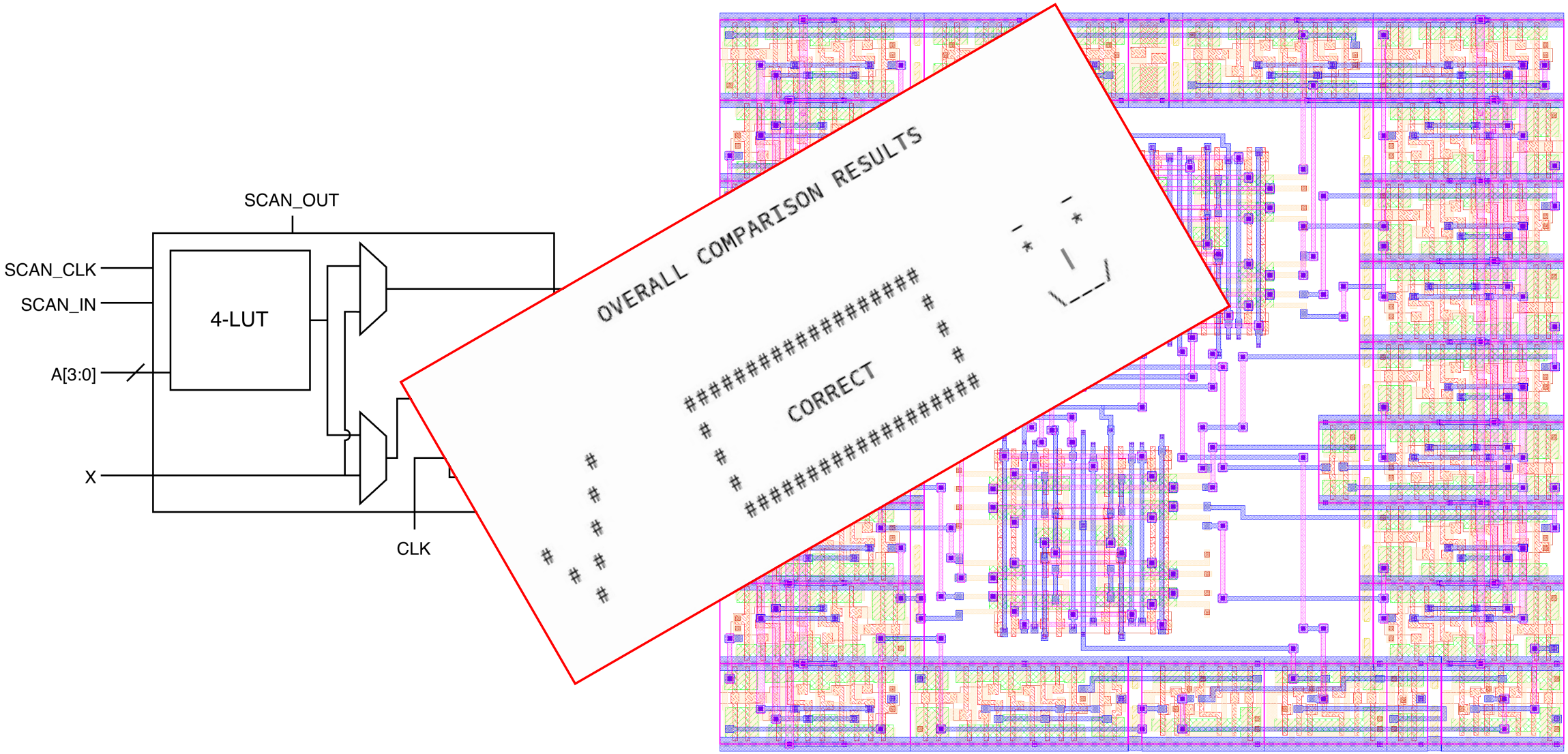
```
dfxtp_params = {
    .height_nm = 2720,
    .width_nm = 460,
    .input_clock_buffer = false,
    .add_inverted_output_port = true,
    .nfet_0_width_nm = 420,
    .nfet_1_width_nm = 360,
    ...
};
```

Parameterised FPGA IP (sky130)

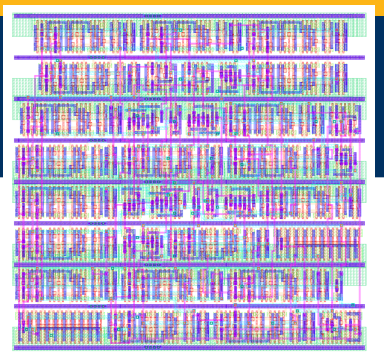
Transmission Gate Stack Generator

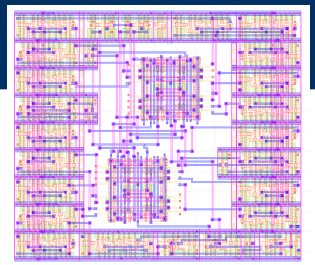


4-LUT Configurable Logic Block



Experimental Results



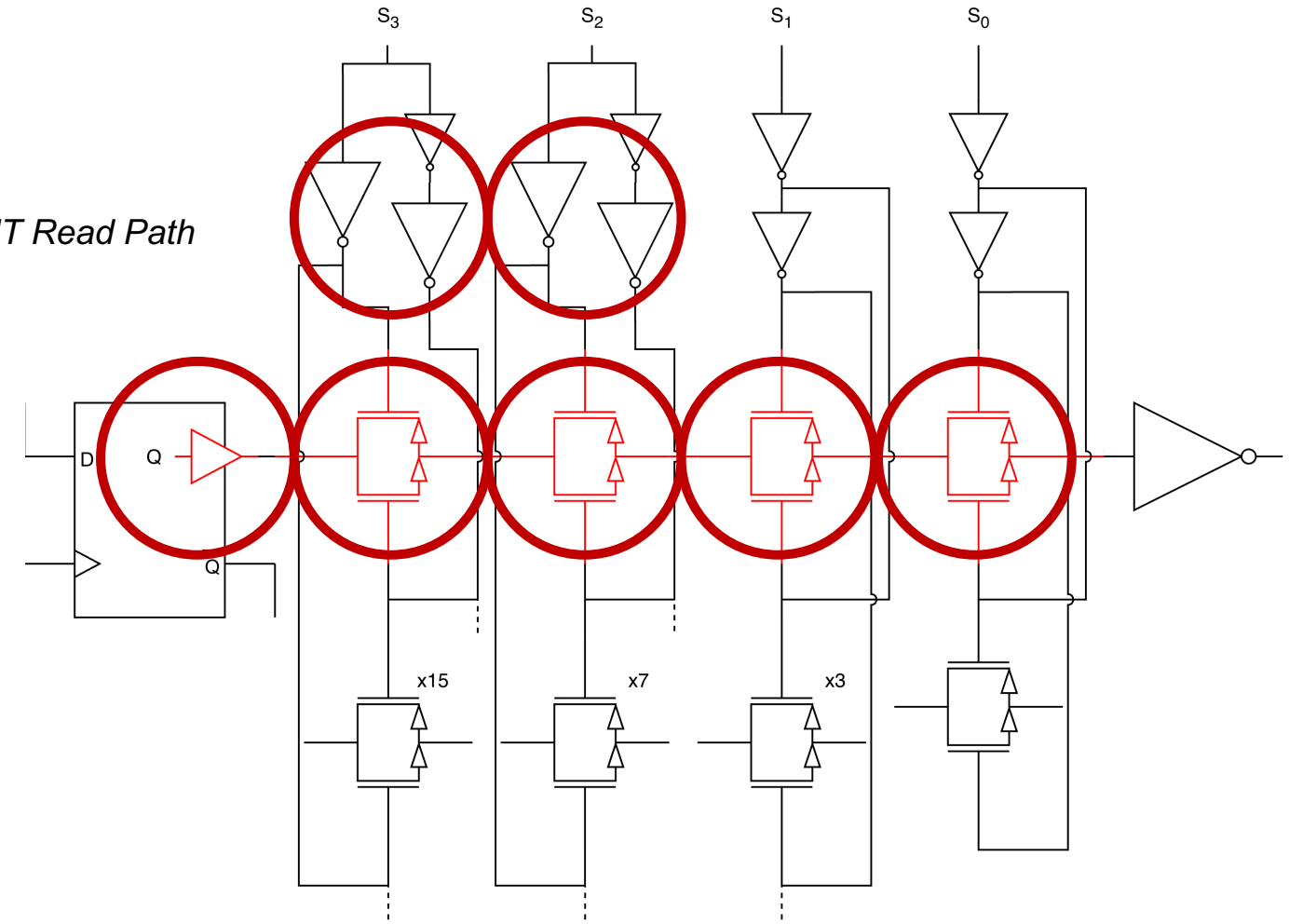


(To scale.)

Metric	LibreLane (OpenROAD) HD Library	Innovus HD Library	Innovus SCL Library	BFG	BFG vs. best *working*
PAR Density (%)	81.6	99.8	96.6	😎	😎
Area (μm^2)	1506.4	709.4	1159.2	698.9	-39.7%
Max delay (ps) (no parasitics)	1120.8	610.0	540.1	437.5	-19.0%
Max delay (ps) (with parasitics)	1925.2	👎	955.6	867.1	-9.26%
Avg power (µW) (with parasitics)	319.7	👎	242.5	183.6	-24.3%
Norm. Power, Delay, Area Product	**3.45**	👎	**1.00**	**0.414**	**-58.6%**

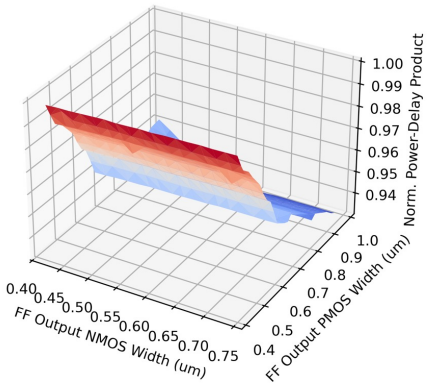
Circuit Tuning

The LUT Read Path

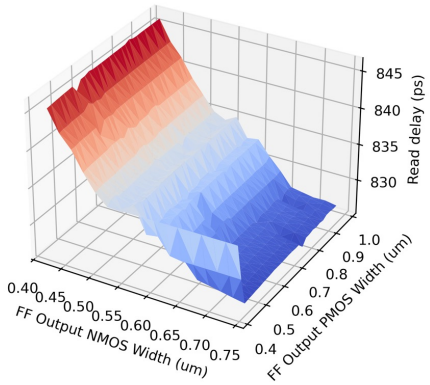


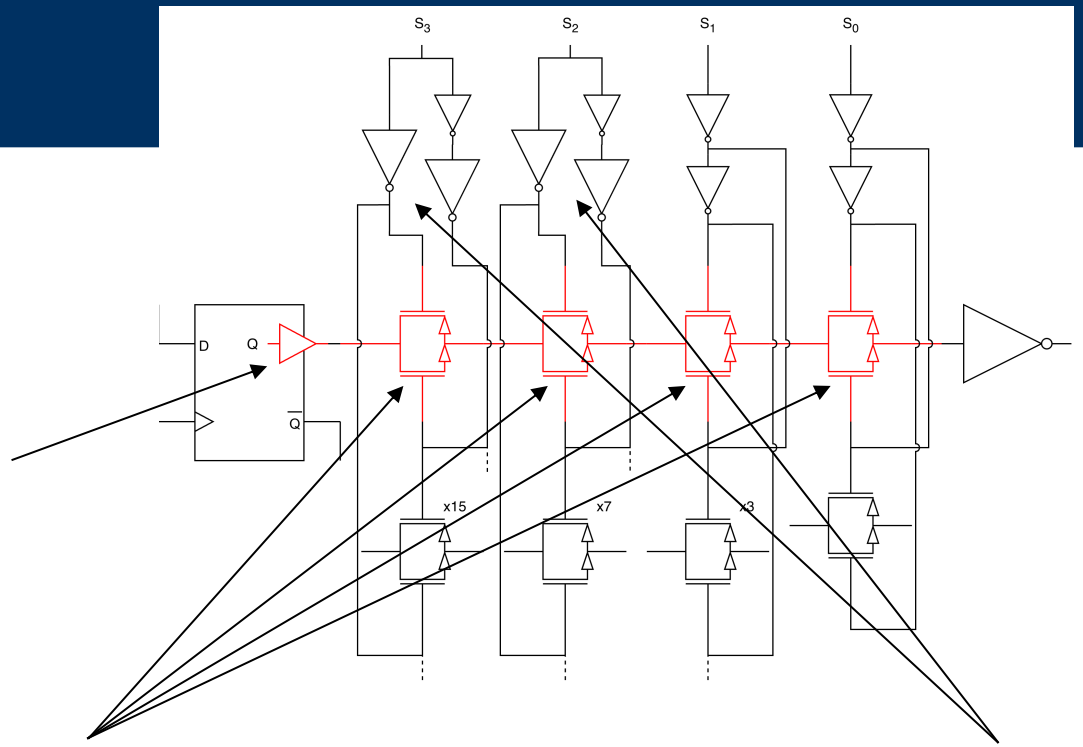
Circuit Tuning

Power-delay product vs
flip-flop buffer sizing

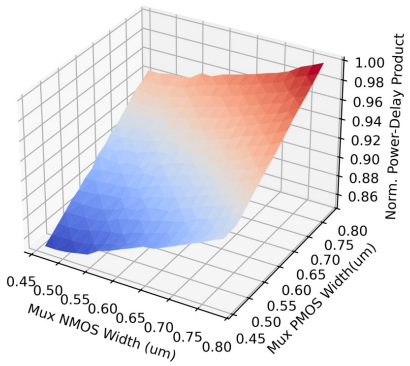


Delay vs.
flip-flop buffer sizing

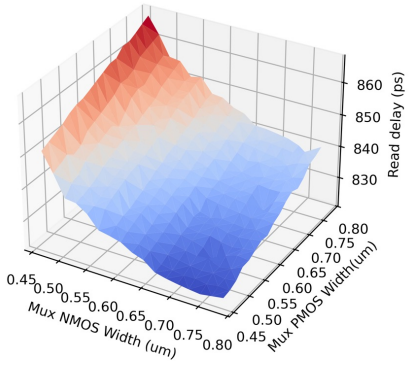




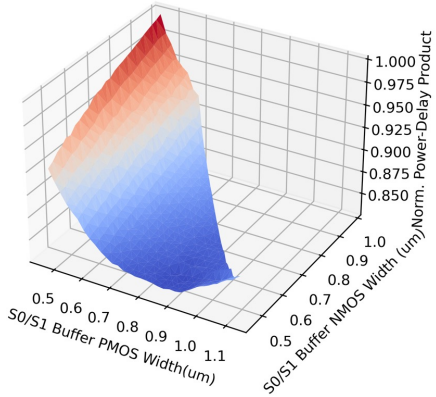
Power-delay product vs
transmission gate sizing



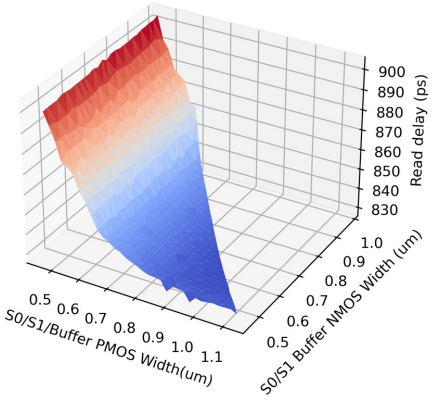
Delay vs.
transmission gate sizing



Power-delay product vs
transmission gate sizing

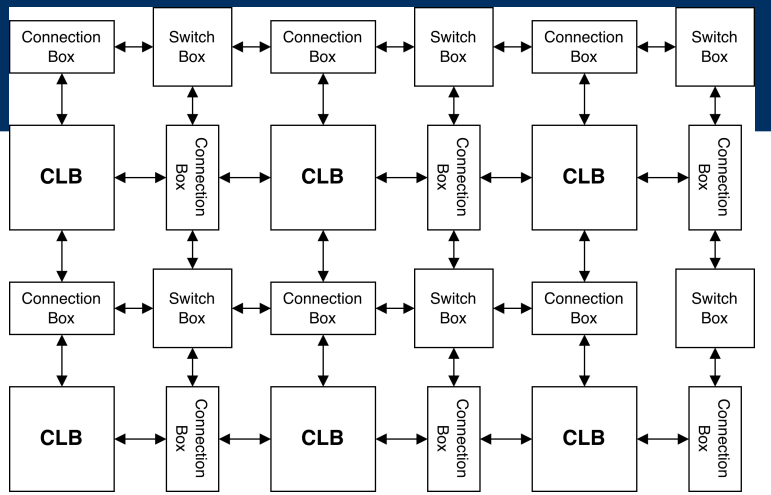


Delay vs.
transmission gate sizing



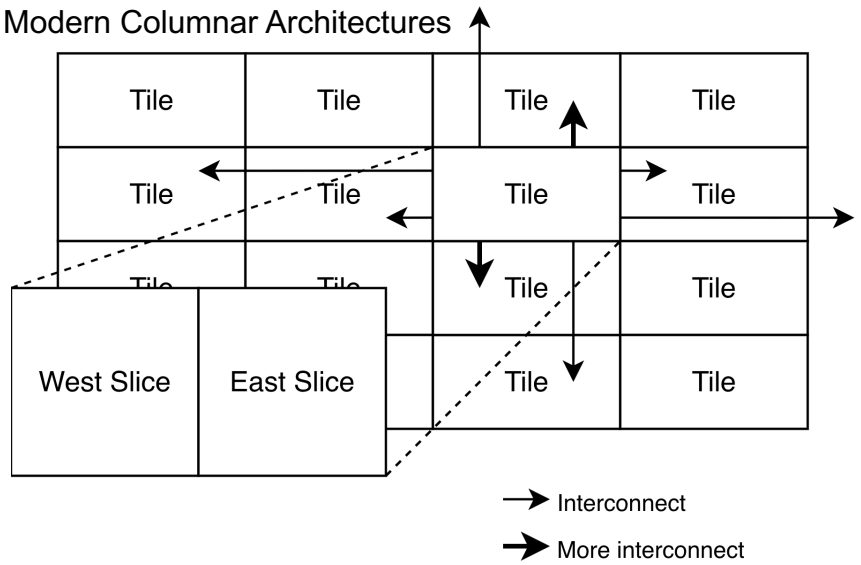
UltraScale-like Interconnect Generator

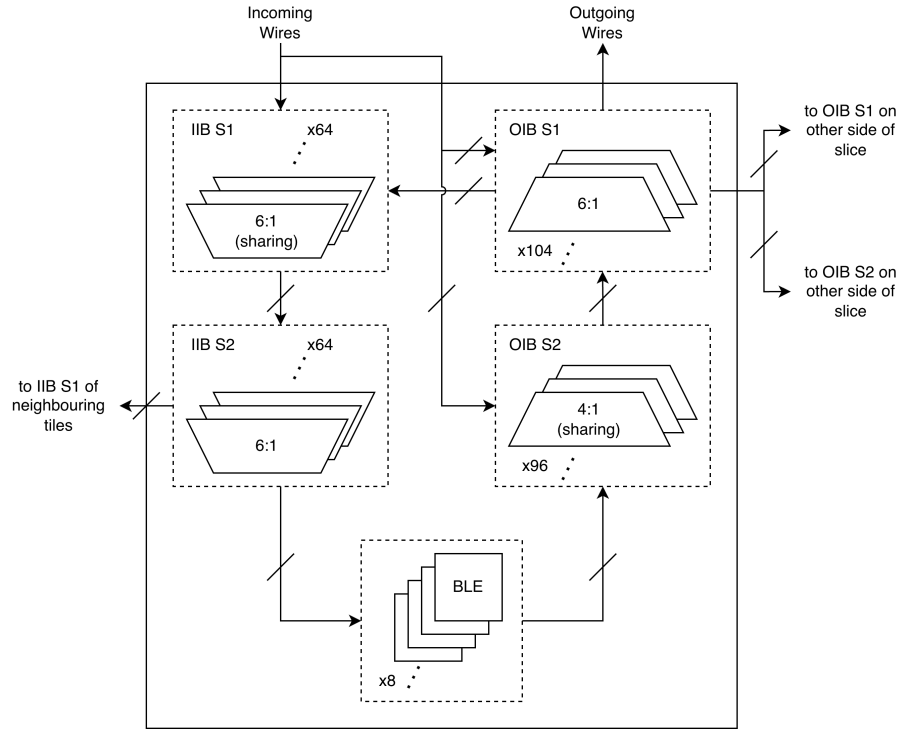
- Modern fabrics depart from academic “island-style” norm with columnar structure
- 2-fly butterfly-ish input/output connection blocks
- Algorithms generalise* fan-out patterns
- Python script produces netlist that can be routed by BFG router
- Feasible!



Traditional “Island-Style” Architectures

VS.





*from one example

A Columnar Architecture By BFG

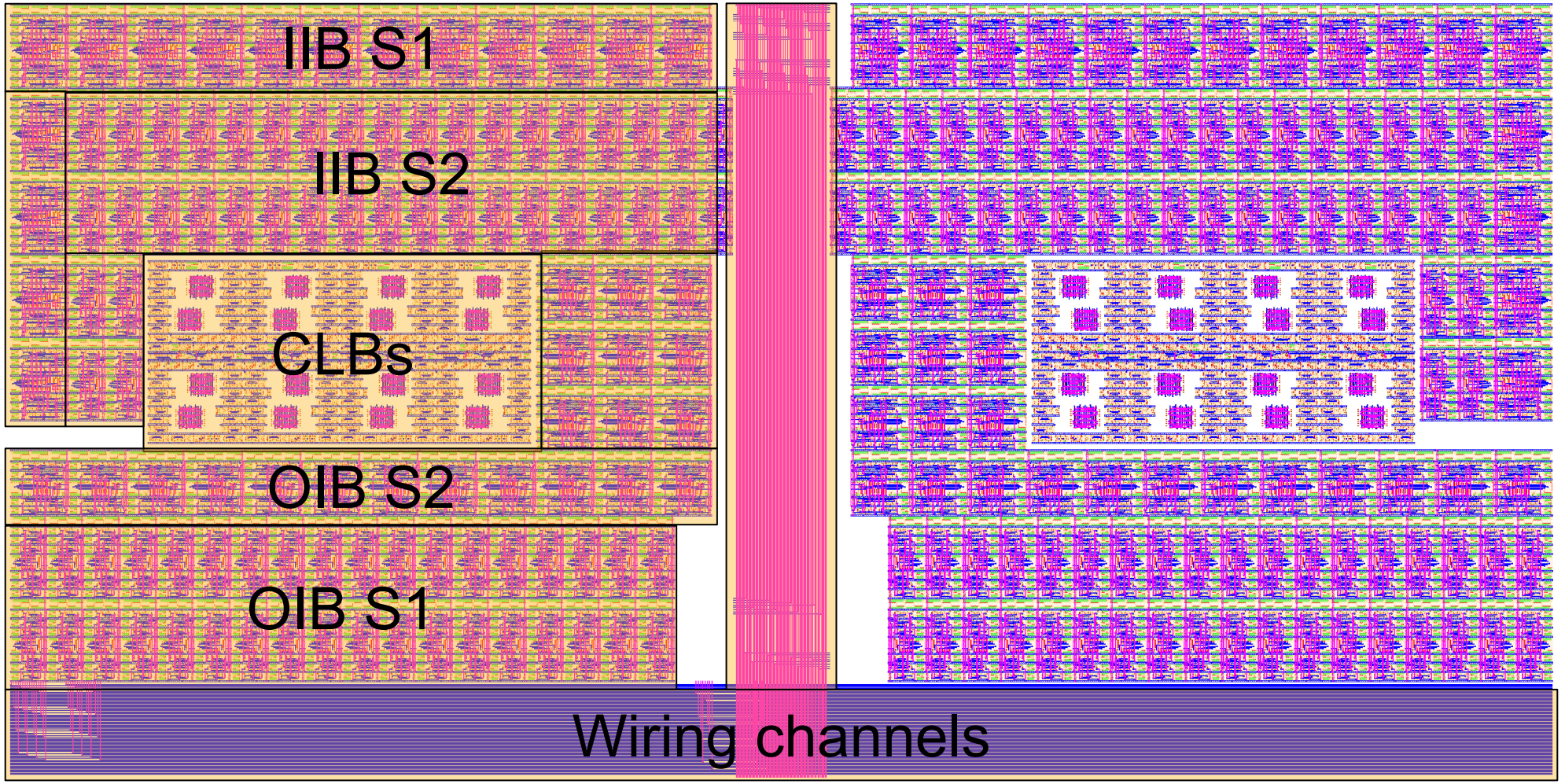
32x 7:2 Muxes

48x 6:1 Muxes

4x S44s

24x 5:2 Muxes

36x 6:1 Muxes



Length 1, 2, 6 wires

4 wires per bundle

Future Work

1. **Improve robustness, CI**
2. **Fix some glaring bugs**
3. **Scale router up and down**
   - Simpler model for large problems
   - Meta optimization loop (rip up and replace, etc)
   - ILP, QLP solvers for small problems
4. **Clock tree, global reset**
5. **Tape-out**
6. **Python-wrap library**
   - More approachable for iterating on layout recipes
7. **ML/AI integration**
   - Expose API
8. **A simple GUI for designers**
   - Debug
   - Guide writing layout recipes
9. **Constraint solvers for parts of layout**
10. **FinFET and newer Process IP**
11. **Integrate with other AMS frameworks**
    - There are now several
12. **A simple, expressive DSL for layout**

Conclusion and Call for Contributions

BFG:

1. **is a C++ framework for analog/mixed-signal design, `libbfg`**
   - geometry, netlist, manual placement & automatic routing
   - purely FOSS dependencies
2. **has a collection of generators ("p-cells") for making decent FPGA IP**
   - CLB achieves 59% better PPA than standard cell implementation
3. **demonstrates the feasibility of building a modern FPGA fabric with a FOSS silicon compiler**

Contributions are welcome!

- **BFG provides something for the community to iterate on**
  - A repository for circuit and layout know-how
- **Layout is hard, but worth it**

Acknowledgements

Collaborators and contributors:

- Berkeley Wireless Research Centre
- John Wawrzynek
- Jonathan Greene
- Eddie Hung
- Dirk Koch

Invaluable tools:

- kLayout, magic, Xyce

Funding:

- Google, Huawei, NSF

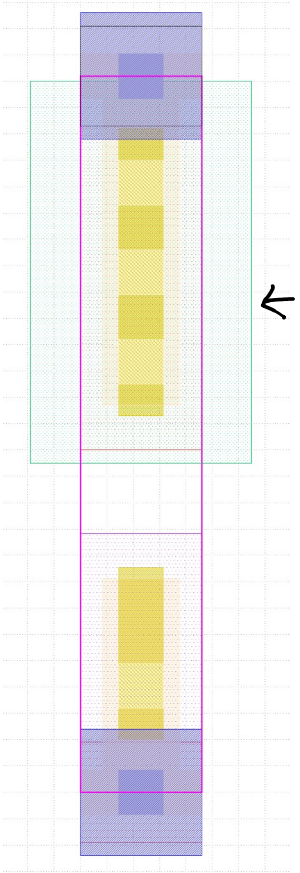
Berkeley
Wireless Research Center

Questions?

Berkeley
Wireless Research Center

Backmatter

Parameterised FPGA IP (sky130)



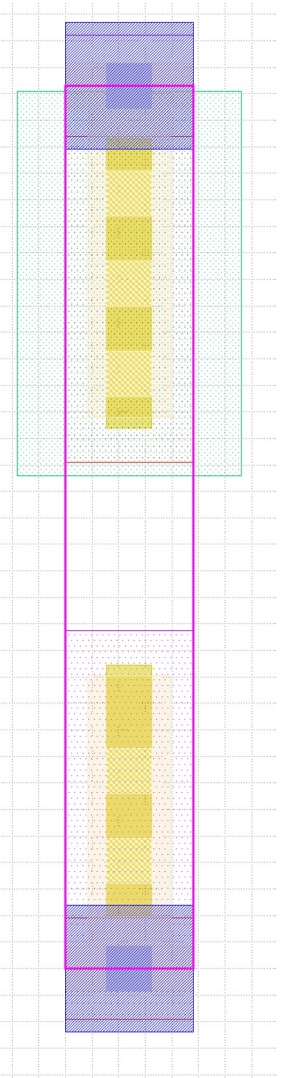
```
tap_params = {
    .height_nm = 2720,
    .width_nm = 460
};
```

matches
sky130_fd_sc_**hd**__tapvpwrvgnd

Tap Cell Generator

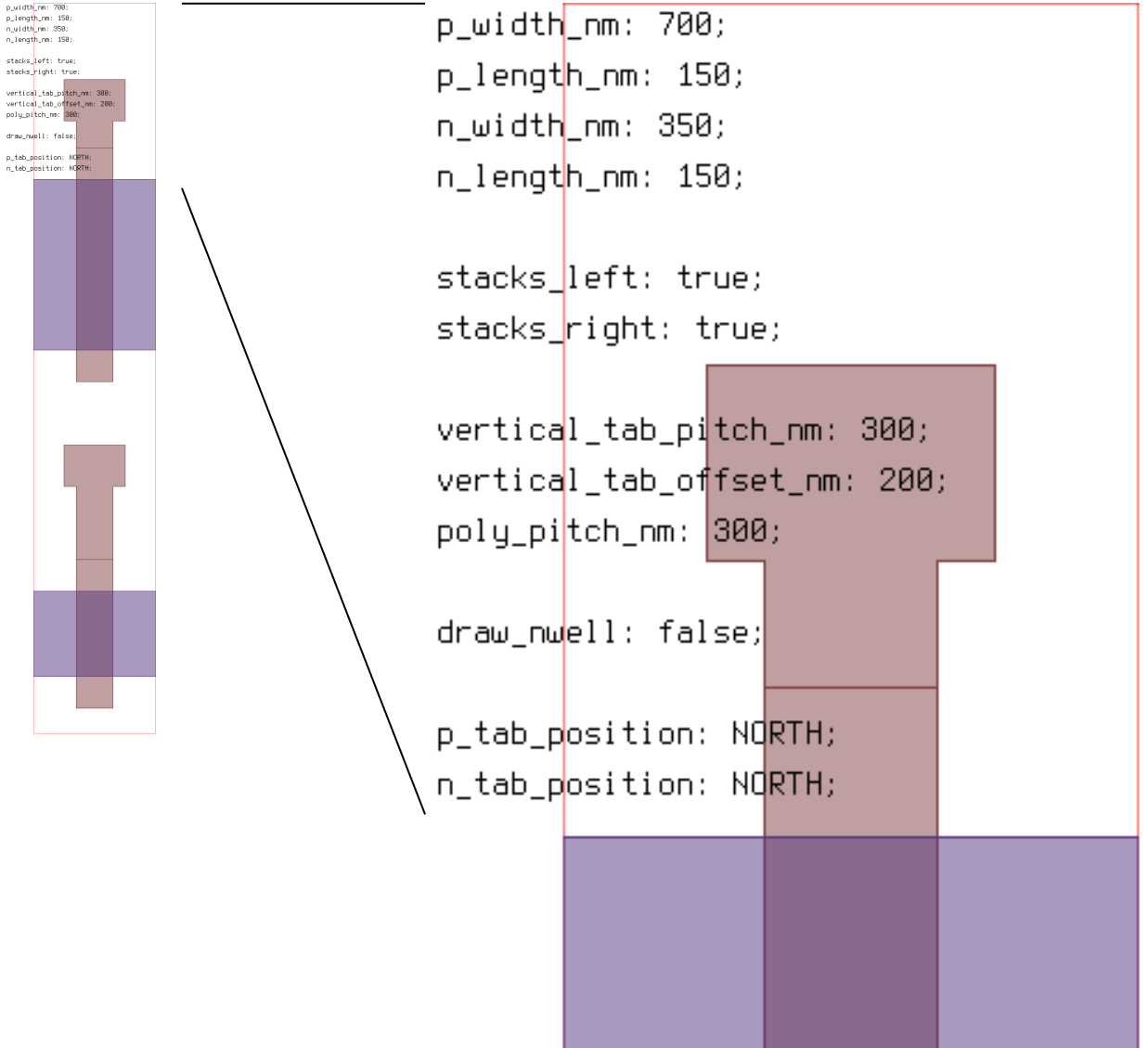
```
tap_params = {
    .height_nm = 3330,
    .width_nm = 480
};
```

matches
sky130_fd_sc_**hs**__tapvpwrvgnd

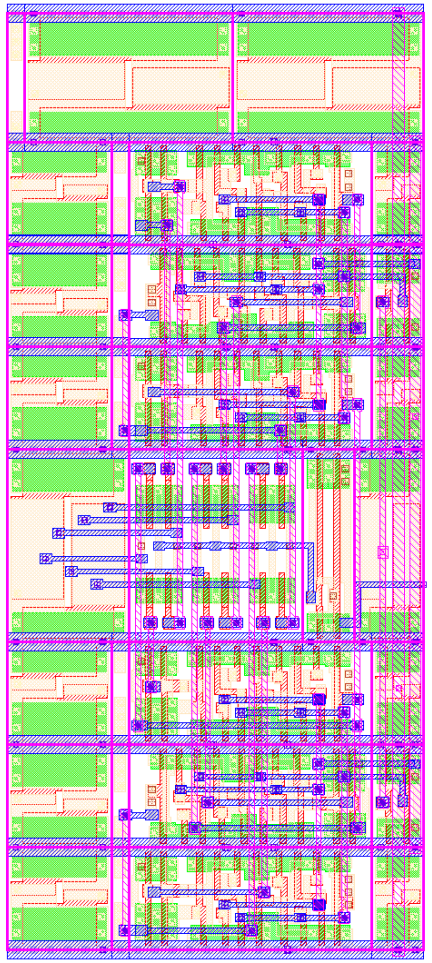


Parameterised FPGA IP (sky130)

Transmission Gate Generator



One-Hot Transmission-Gate Muxes



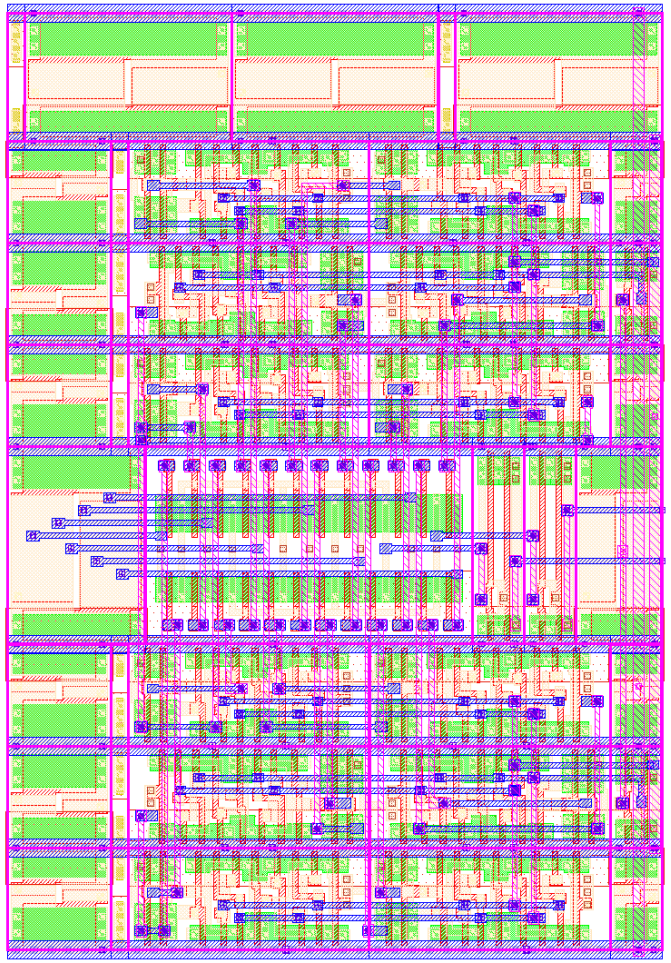
config struct

```
num_inputs: 6
num_outputs: 1
```

6:1

config struct

```
num_inputs: 7
num_outputs: 2
```



Two 6:1s sharing 5 inputs each
“7:2”

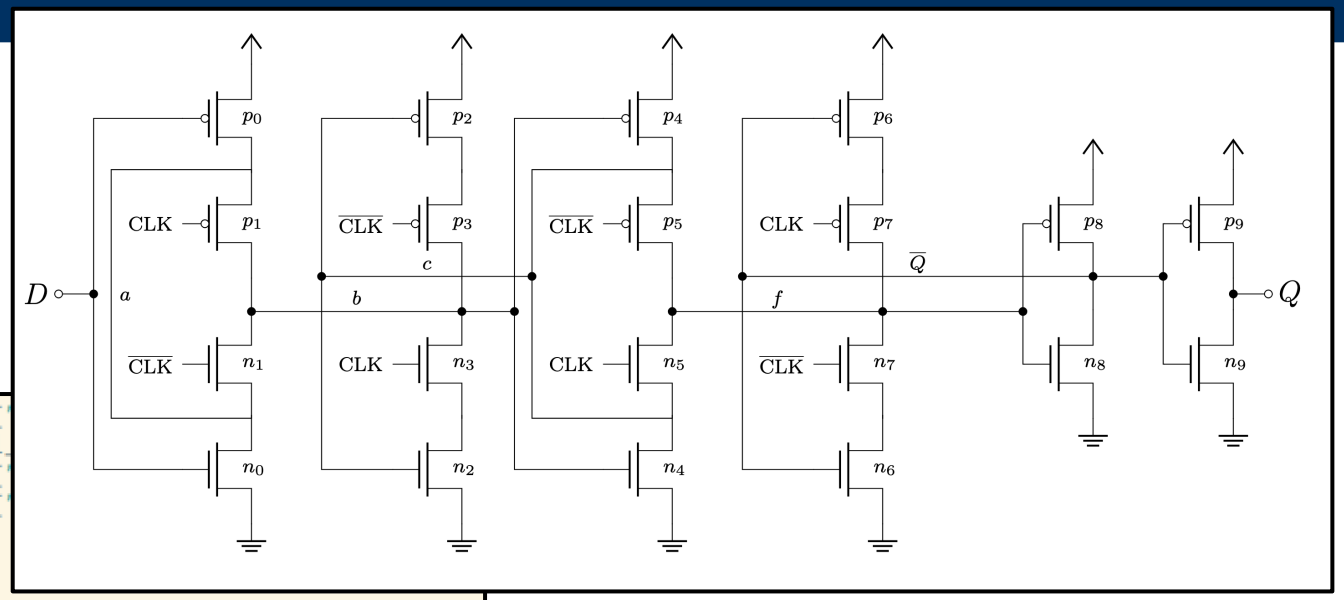
Circuit (Netlist)

Assemble a circuit while drawing it:

```
97  circuit::Instance *nfet_8 = circuit->AddInstance("nfet_8", nfet_01v8);
98  circuit::Instance *nfet_9 = circuit->AddInstance("nfet_9", nfet_01v8);
99
100 // These are all sky130_fd_pr__pfet_01v8_hvt:
101 Circuit *pfet_01v8_hvt = design_db_->FindCellOrDie(
102     "sky130", "sky130_fd_pr__pfet_01v8_hvt")->circuit();
103 circuit::Instance *pfet_0 = circuit->AddInstance("pfet_0", pfet_01v8_hvt);
104 circuit::Instanc
105 circuit::Instanc
106 circuit::Instanc
107 circuit::Instanc
108 circuit::Instanc
```

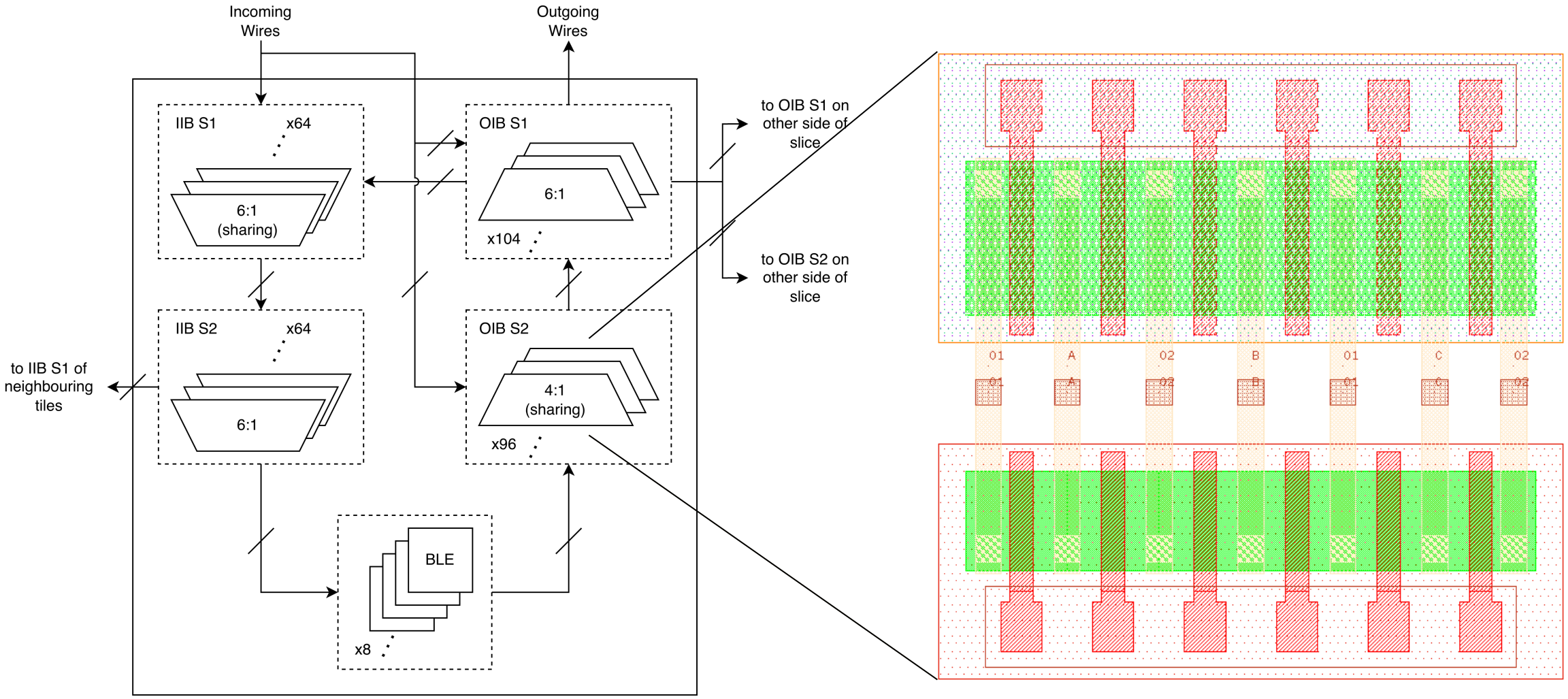
```
338  pfet_0->Connect({{"d", a}, {"g", D}, {"s", VPWR}, {"
339  pfet_1->Connect({{"d", b}, {"g", effective_internal_
340  nfet_1->Connect({{"d", b}, {"g", CLKI}, {"s", a}, {"
341  nfet_0->Connect({{"d", a}, {"g", D}, {"s", VGND}, {"
342
343  circuit::Wire c = circuit->AddSignal("c");
344  circuit::Wire j = circuit->AddSignal("j");
345  circuit::Wire e = circuit->AddSignal("e");
346
347  pfet_2->Connect(
348  pfet_3->Connect(
349  nfet_3->Connect(
350  nfet_2->Connect(
```

```
376 circuit::Wire f = circuit->AddSignal("f");
377
378 pfet_4->Connect({{"d", c}, {"g", b}, {"s", VPWR}, {"b", VPB}});
379 pfet_5->Connect({{"d", f}, {"g", CLKI}, {"s", c}, {"b", VPB}});
380 nfet_5->Connect({{"d", f}, {"g", effective_internal_clock}, {"s", c}, {"b", VNB}});
381 nfet_4->Connect({{"d", c}, {"g", b}, {"s", VGND}, {"b", VNB}});
382
383 circuit::Wire Q_B = circuit->AddSignal("Q_B");
384 circuit::Wire h = circuit->AddSignal("h");
385 circuit::Wire i = circuit->AddSignal("i");
386
387 pfet_6->Connect({{"d", h}, {"g", Q_B}, {"s", VPWR}, {"b", VPB}});
388 pfet_7->Connect({{"d", f}, {"g", effective_internal_clock}, {"s", h}, {"b", VPB}});
389 nfet_7->Connect({{"d", f}, {"g", CLKI}, {"s", i}, {"b", VNB}});
390 nfet_6->Connect({{"d", i}, {"g", Q_B}, {"s", VGND}, {"b", VNB}});
```

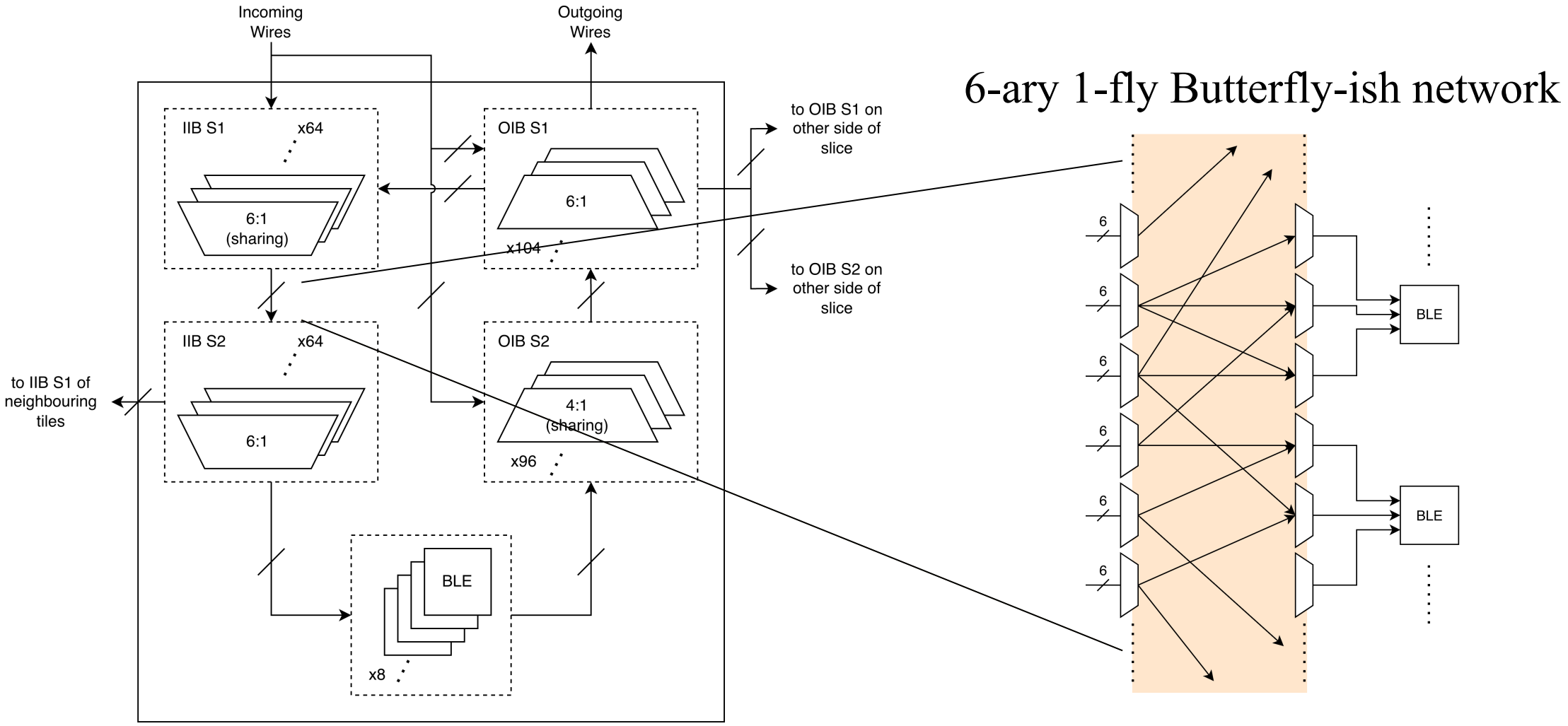


atoms/sky130_dfxtp.cc

UltraScale Interconnect

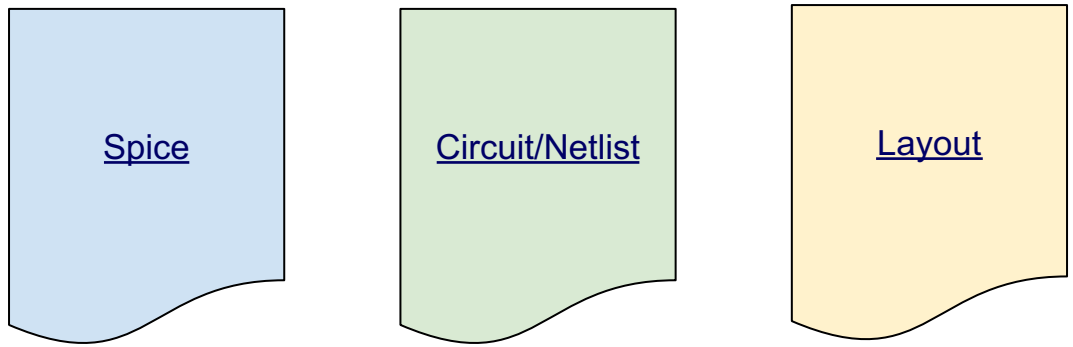


Mysterious Fan-Out Patterns Between Stages



VLSIR

- **Interchange schemas for database information**
  - Dan Fritchman, Aviral Pandey, myself
  - Decouple tools from large, complex programs
  - Unix-like philosophy: many point programs
    - each do a few things well
  - Expressed as Protocol Buffers (or similar)
- **Write whatever modern language you want:**
  - C++, python, Rust, C++, Typescript

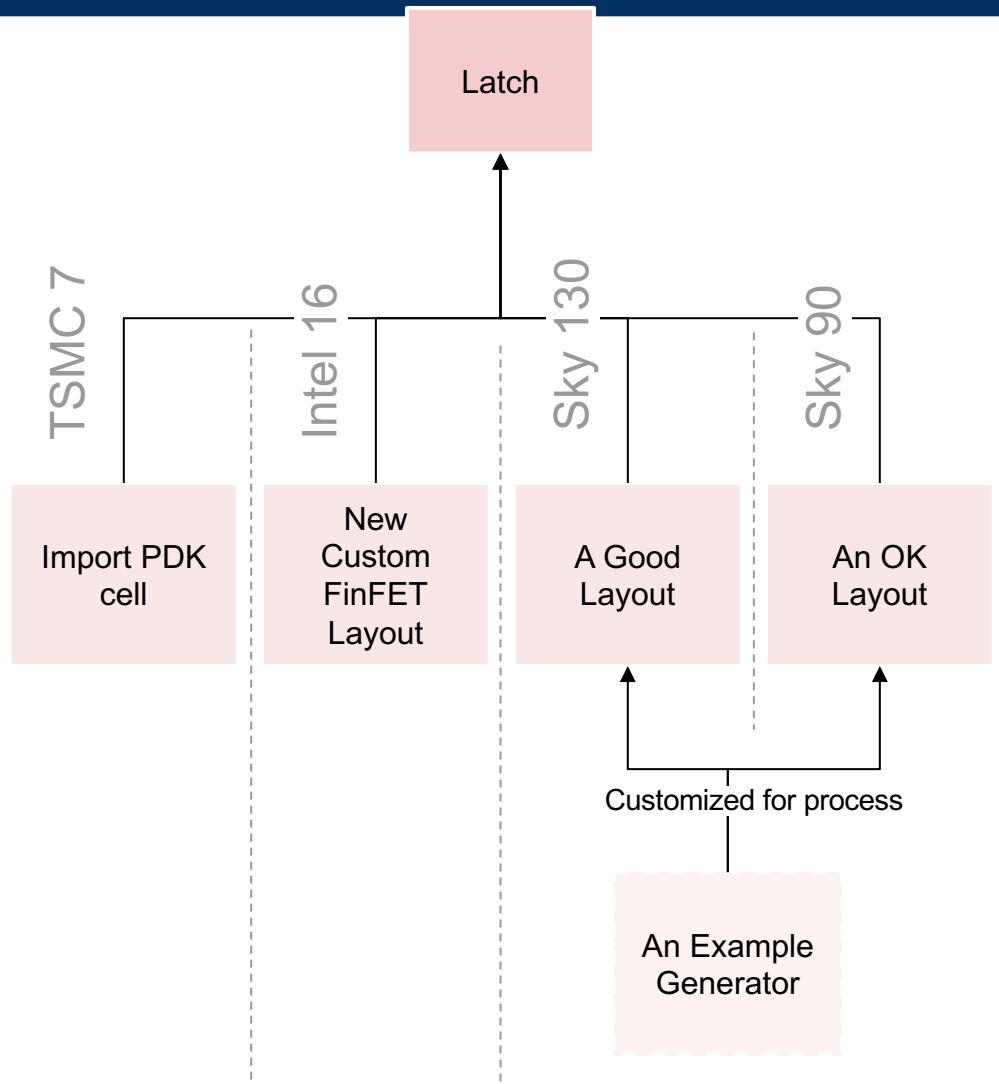


Protocol Buffer Schemas

```
33  message Layer {
34    int64 number = 1;
35    int64 purpose = 2;
36  }
37
38  // # Rectangle Primitive
39  message Rectangle {
40    // Net Name
41    string net = 1;
42
43    // The lower-left corner of th
44    Point lower_left = 2;
45    int64 width = 3;
46    int64 height = 4;
47  }
48
49  // # Polygon Primitive
50  message Polygon {
51    // Net Name
52    string net = 1;
53
54    // List of Vertices
55    // `Polygons` implicitly "clos
56    // N-sided `Polygons` therefor
57    repeated Point vertices = 2;
58  }
```

Building cells process-portably

- **Some families of PDK support similar layout topologies (e.g. sky130, gf180)**
  - Same code, different numbers
- **Some require totally different layouts**
  - Can't split poly, pitches must be matched, etc
- **Can use PDK std. cells as template**
  - Import and parameterise
  - Only need a subset
- **Allow process-specific customization if needed**
  - But try to avoid



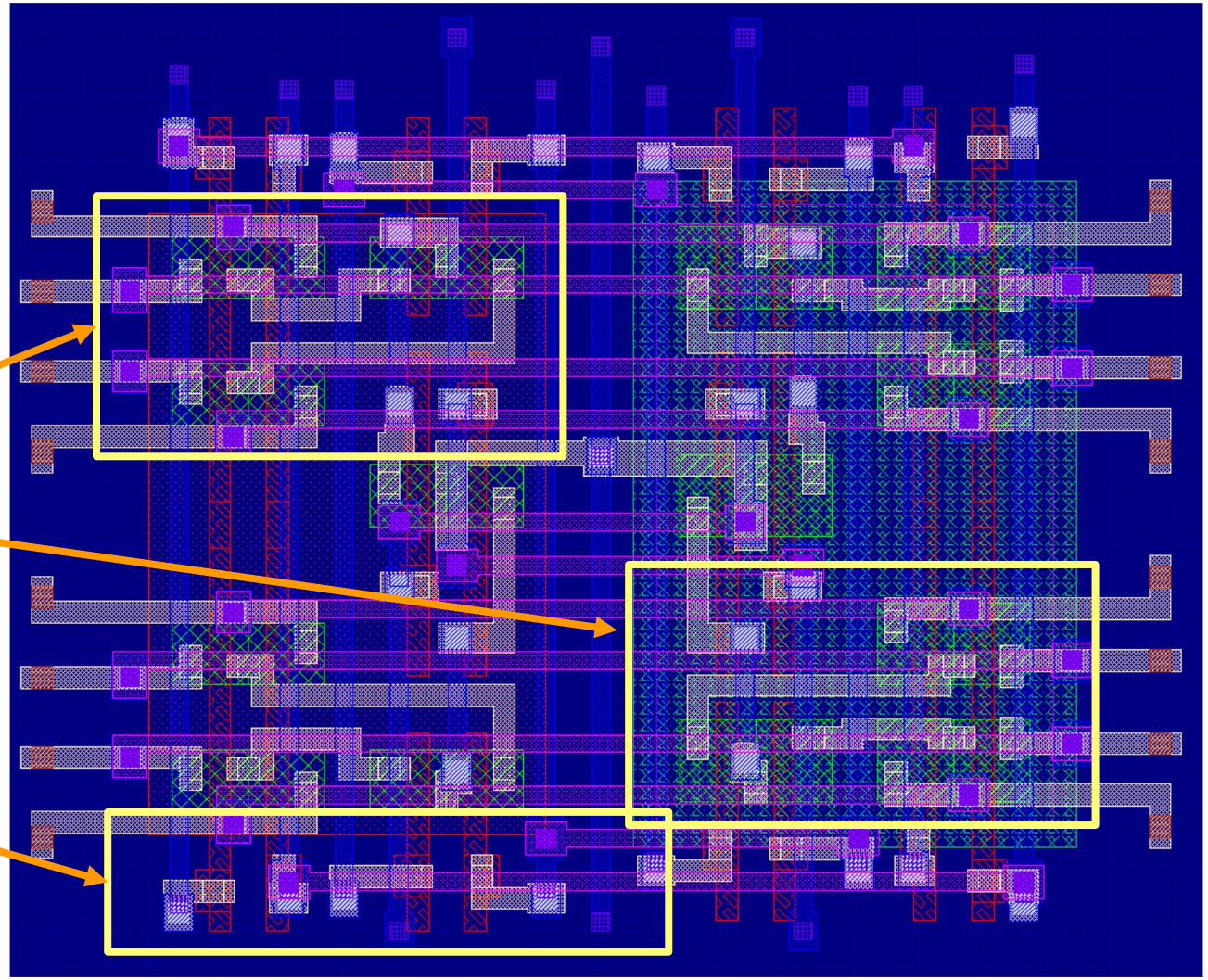
Process Portability: sky130_mux

One function generates the first level (6 transistors)

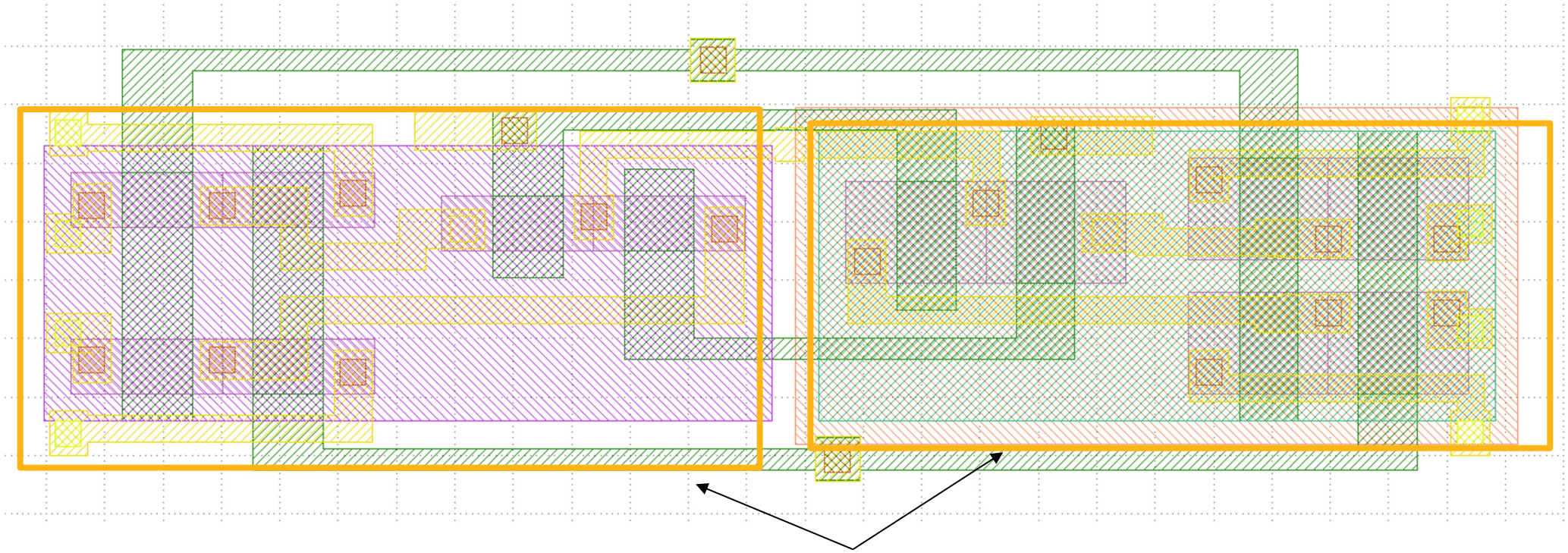
Called 4 times: layout is flipped horizontally & vertically to create other quadrants

Unique transistor sizes per path (esp. P- vs NMOS)

Wires, poly, ports are then placed meticulously

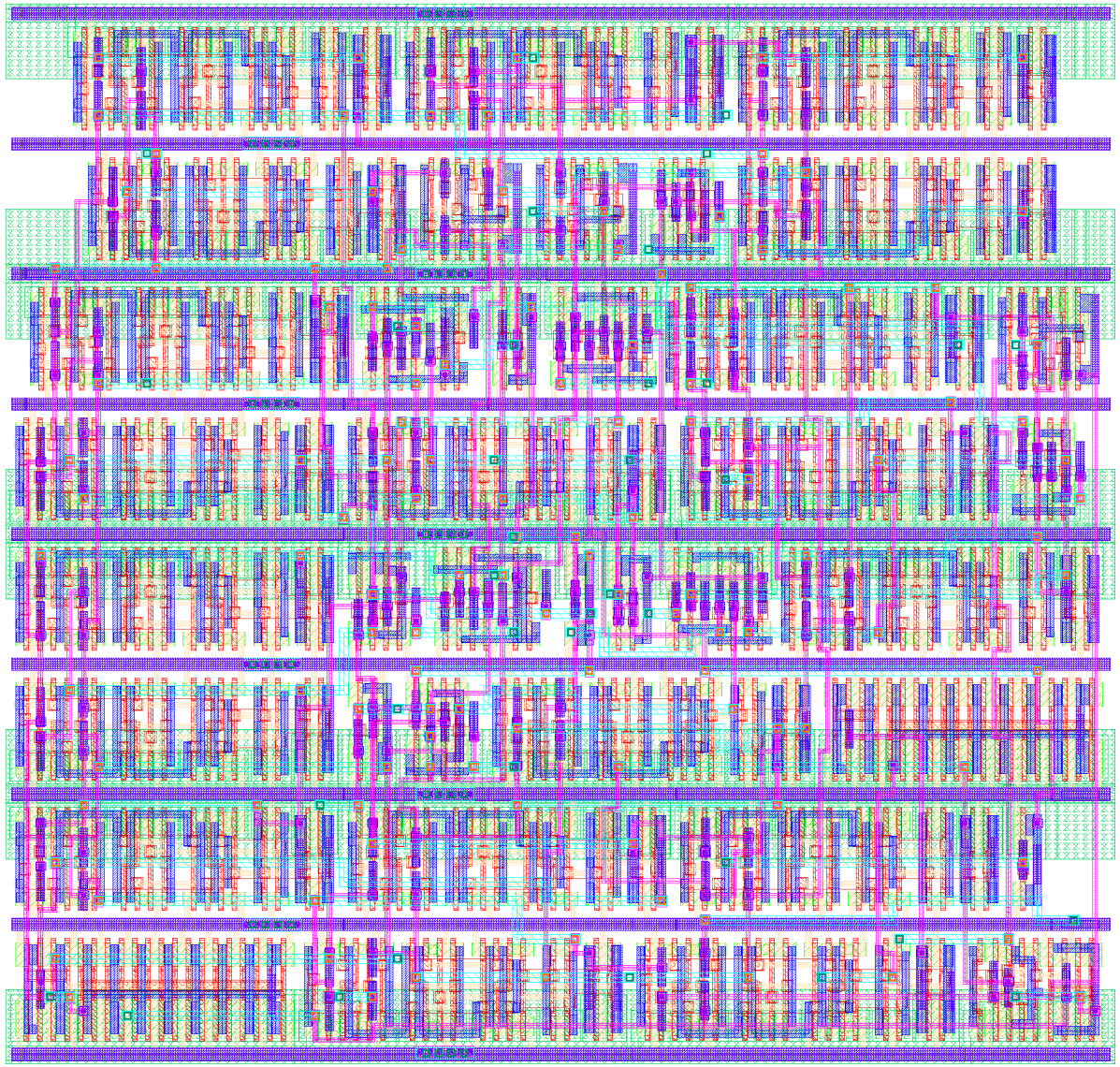


Process Portability: gf180mcu_mux4



Same generating function as in the sky130 8-MUX, different parameters

4-LUT Configurable Logic Block in Sky130 SCL



The Tile Interface

```
class Tile {
 public:
  Tile(DesignDatabase *design_db)
      : design_db_(design_db) {}

  virtual bfg::Cell *Generate() = 0;

  // Some other helpful methods

 protected:
  std::string name_;
  DesignDatabase *design_db_;
};

}  // namespace atoms
}  // namespace bfg
```

Commercial eFPGAs

- Achronix, Adicsys, Efinix, Flex Logix, Menta, QuickLogic
- Gartner expected market share of semiconductors with eFPGA to approach $10B in 2023