UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title

BFG: An Open-Source Silicon Compiler for High-Performance Reconfigurable Fabrics

Permalink

https://escholarship.org/uc/item/52k1w8vx

ISBN

9798173786791

Author

Reais-Parsi, Arya Bahram

Publication Date

2026-08-10

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

BFG: An Open-Source Silicon Compiler for High-Performance Reconfigurable Fabrics

by

Arya Bahram Reais-Parsi

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering & Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor John Wawrzynek, Chair
Professor Borivoje Nikolic
Professor Sophia Shao
Jonathan Greene

Summer 2026

BFG: An Open-Source Silicon Compiler for High-Performance Reconfigurable Fabrics

Copyright 2026
by
Arya Bahram Reais-Parsi

Abstract

BFG: An Open-Source Silicon Compiler for High-Performance Reconfigurable Fabrics

by

Arya Bahram Reais-Parsi

Doctor of Philosophy in Electrical Engineering & Computer Sciences

University of California, Berkeley

Professor John Wawrzynek, Chair

Synthesisable eFPGA generators have enabled considerable innovation and access to FPGA architecture research in recent years. By generating RTL descriptions of FPGA fabrics, these tools leverage established digital design flows to meaningfully reduce the time taken to tape-out a validated FPGA fabric. Unfortunately, the convenience of employing standard-cells over custom layout comes at the cost of reduced circuit performance. Research architectures lag their commercial counterparts in terms of performance, power and area. At the same time, the end of Moore's Law and Dennard Scaling is contributing to unprecedented demand for domain-specific accelerators. Performant, easy-to-use and cheaper reconfigurable architectures are needed from the research community to meet this demand. This work presents BFG, an alternative approach to FPGA generation that programmatically generates all circuits together with custom layout. In SkyWater 130 nm, BFG's cells achieve up to 59% lower power-delay-area product than equivalent logic synthesized with standard cells. In this work we present each of BFG's four contributions: an unencumbered open-source C++ AMS design framework; a collection of parametric IP generators for FPGA components; a study reverse-engineering a modern commercial architecture; and, a modern columnar fabric architecture together with an example of how BFG might be used to build it. Though our study reveals significant complexity to this approach, we conclude that the performance benefits are justified by the shared development model. BFG provides researchers, enthusiasts and entrepreneurs a repository of fast circuitry and corresponding layout for the development of better FPGA IP.

In loving memory

Ladan Zolghadri

Bob Brodersen

Myoho

Dedicated to Cal Ski Club

... and also my parents,

Lobat & Mehrdad Reais-Parsi

Acknowledgments

I owe a debt of gratitude to all of the individuals and organisations that have supported me through this work. But, for first and last time in this document, I will try to brief. Thanks first of all to Prof. John Wawrzynek: advisor, drinking buddy, golf buddy, friend. Thank you to my collaborators Dan Fritchman, Aviral Pandey, Jonathan Greene, Eddie Hung, Alan Mishchenko, Dirk Koch, Tim Ansell, Rohan Kumar and Rahul Kumar for your patience, wisdom and humour. Thank you to my dissertation committee, Prof. Borivoje Nikolic and Prof. Sophia Shao, for your support whenever it was needed. Thank you to Prof. Scott Shenker, Prof. Satish Rao and Prof. Bob Brodersen for the mentorship and the guidance, friendship and support. A big thank you to Prof. Ali Niknejad for the back-up when I needed it, and to Prof. Krste Asanović and Prof. David Patterson for the inspiration to come to Berkeley.

Special thanks to Xili Mo, who is probably the second and last person to ever read every word of this document. Thanks to the friends who've never brought up how long I've been a student, Sam Hillman, Zach Beardman, Jazmyne Dodd, Kenny Alyce; to Richard Alexander, who offered to read this not knowing what he would be getting himself into; to my friends Emily Marx, Mauricio Bustamante, Sarah McClure, James Martin, Alon Amid, Miles Rusch and Andrew Waterman, for sticking with me through life's vicissitudes; and thank you to Adam Lagan and Liam Davies for teaching me about sports. To Nana Pat, thank you from the bottom of my heart for your tireless support from afar. Thank you to Zoe and Sara Cohen; to my friends in the ADEPT/SLICE/BLAST lab; and to my friends from BWRC. And thank you to everyone I co-founded a startup with.

A huge thank you to my friends in the Cal Ski Club for the strength you daily gave me to make it to the next cabin trip; summer was never the same after you left. Thank you to Sophia Raab for sharing culture and for sharing Myoho[1] with me, and thank you to Myoho for your joyous little heart. Thank you Heather[1], Steve[1] and Iman[1] for every time you chose to curl up on me over everyone else. And finally, thank you to my parents, Lobat and Mehrdad Reais-Parsi, for the endless love and support you have given me my whole life.

[1] These are cats.

Contents

Contents **v**

List of Figures **vii**

List of Tables **xi**

1 Introduction **1**
1.1 Some History . . . 1
1.2 A New Golden Age of Computer Architecture . . . 3
1.3 Electronic Design Automation . . . 4
1.4 Field Programmable Gate Arrays . . . 10
1.5 Motivation . . . 12
1.6 Aims and Hypothesis . . . 15
1.7 Related Work . . . 17
1.8 Contributions . . . 24
1.9 Outline . . . 25

2 BFG **27**
2.1 An Analog/Mixed-Signal Generator Framework . . . 27
2.2 Data Model . . . 28
2.3 Geometric Routines . . . 44
2.4 Process Portability . . . 72
2.5 Creating Circuits . . . 73
2.6 Creating Layout . . . 77
2.7 Automatic Routing . . . 80
2.8 Ecosystem Integration . . . 94
2.9 Microservices for VLSI . . . 99

3 Interconnect **103**
3.1 Xilinx UltraScale . . . 103
3.2 Methodology . . . 105
3.3 Interconnect Wiring . . . 117
3.4 Input Interconnect Block . . . 117

3.5 Output Interconnect Block 140
3.6 Interconnect Crossbar 160
3.7 Control Paths 163

4 Parametrised Cell Generators for FPGAs **167**
4.1 Paradigms 167
4.2 Usage 169
4.3 Basic Elements 171
4.4 4-LUT CLB 201
4.5 Carry Chain Bit 218
4.6 S-44 CLB 221
4.7 $N:1$ Interconnect Multiplexer 224
4.8 $M:2$ Interconnect Multiplexer 242
4.9 Interconnect Wire Block 246
4.10 Unnamed Columnar Architecture 250

5 Discussion **258**
5.1 Future Work 261

Bibliography **274**

A Source Code **285**

List of Figures

1.1 Traditional island-style FPGA architectures . . . 11
1.2 So you want to compute with FPGAs? . . . 13
1.3 Desired BFG inputs and outputs . . . 16
1.4 SkyWater 130 nm Process Stack Diagram . . . 24
1.5 Global Foundries 180 nm Process Stack Diagram . . . 25

2.1 Some intra-layer rules . . . 32
2.2 Some inter-layer rules . . . 33
2.3 Non-commutative DRC rules . . . 34
2.4 Synthesised rules: via encapsulation . . . 35
2.5 Overview of the flattening operation provided by `Circuit` class . . . 39
2.6 Geometric types . . . 40
2.7 Brute force checking for the distance between rectangles . . . 46
2.8 BFG `Lines` are both bounded and unbounded, depending on who's asking . . . 48
2.9 BFG `Arcs` and whether or not they intersect points . . . 49
2.10 Casting a ray to test point-polygon intersection . . . 50
2.11 Coincidence of lines with the edges of non-convex polygons . . . 51
2.12 A line intersecting an arc . . . 52
2.13 Intersections between rectangles . . . 54
2.14 Intersections between rectangles and polygons . . . 55
2.15 Ray-casting to find the intersection between rectangles and polygons . . . 55
2.16 A rounded rectangle and its regions . . . 58
2.17 Optimistic cartoon of a wire path and its resulting polygon . . . 59
2.18 Examples of notches between non-intersecting wire edges . . . 59
2.19 An example 3-segment wire modeled by a `PolyLine` . . . 61
2.20 Generated boundary lines for segments of a `PolyLine` . . . 62
2.21 Procedure for inflating a `PolyLine` into a `Polygon` . . . 64
2.22 `PolyLine` complications . . . 66
2.23 Row management with the `RowGuide` helper . . . 67
2.24 Multiple rows are managed together by the `MemoryBank` helper . . . 68
2.25 Simple row alignment with `MemoryBank` . . . 69
2.26 Checkerboard-style grid management with the `CheckerboardGuide` . . . 71
2.27 An XOR2 circuit . . . 73

2.28 Plan for a transistor layout . . . 77
2.29 Exported GDS of the demonstration transistor . . . 80
2.30 The `RoutingGrid` model . . . 81
2.31 Obstacle avoidance . . . 83
2.32 Connecting an off-grid vertex to nearby tracks . . . 86
2.33 Connecting an off-grid vertex to the nearest possible point . . . 87
2.34 Post-route optimisation legalises tracks . . . 88
2.35 Pre-export optimisation patches oddities . . . 89
2.36 Bookkeeping required when routing vertices and edges are used . . . 90
2.37 Routed wires exported as layout . . . 91

3.1 The Basic Logic Element of a Xilinx UltraScale is its Configurable Logic Block [4] 104
3.2 Slices and tiles in the UltraScale fabric . . . 105
3.3 Overview of the interconnect topology within an UltraScale Slice . . . 106
3.4 Nodes and their names in the RapidWright model . . . 107
3.5 Example output of the `fiddle.py` tool . . . 109
3.6 Example of compound queries with the `fiddle.py` tool . . . 109
3.7 Inputs (1-6, X, I) to a BLE in the UltraScale . . . 111
3.8 Outputs (MUX, O, Q, Q2) from a BLE in the UltraScale . . . 112
3.9 UltraScale tile adjacency matrix . . . 113
3.10 Input sharing across entities distinguished by the second ordinal . . . 115
3.11 Circuit view of two 4:1 transmission-gate multiplexers sharing 3 inputs . . . 116
3.12 Layout view of two 4:1 transmission-gate multiplexers sharing 3 inputs . . . 116
3.13 A non-local bounce node using the other logical side of a regular Stage 1 multiplexer 119
3.14 Interconnect to IIB S1 . . . 124
3.15 IIB S1 to S2 . . . 133
3.16 IIB S1 to S2 but you reorder the drivers . . . 134
3.17 IIB S2 driving the BLE inputs . . . 135
3.18 Bounce muxes drive a local BLE bypass input and neighbouring Stage 1 muxes 136
3.19 TODO: Does this look better than 3.17? . . . 136
3.20 A non-local bounce node using the other logical side of a regular Stage 2 multiplexer 142
3.21 Interconnect to OIB S2 . . . 145
3.22 BLE outputs to OIB S2 . . . 151
3.23 OIB S2 to S1 . . . 159
3.24 Connections between length-12 wires in going the same direction . . . 161
3.25 Eastward length-12 interconnect wires can travel diagonally up and down . . . 161
3.26 Connections between length-12 wires going in different directions . . . 162
3.27 Connectivity between long wires in the UltraScale interconnect . . . 165
3.28 Clock-enable 1 input multiplexer . . . 166

4.1 Buffer circuit . . . 171
4.2 Generated buffer layouts . . . 172

4.3 Split buffer circuit . . . 174
4.4 Layout of the Split Buffer . . . 174
4.5 A flip-flop circuit, without the input clock buffer . . . 177
4.6 Generated flip-flop layouts . . . 178
4.7 Default MOSFET bulk connections by nearby tap cells . . . 179
4.8 Tap cell layout . . . 180
4.9 Decap cell circuit . . . 181
4.10 Generated decap cell layouts in three sizes . . . 182
4.11 Active 2 : 1 multiplexer circuit . . . 188
4.12 Active 2 : 1 multiplexer layout . . . 189
4.13 The `sky130_fd_sc_hd__xor2_1` circuit. . . . 191
4.14 Layout generated by `Sky130Xor2` . . . 191
4.15 Generated layout of the `Sky130Mux` . . . 193
4.16 Composition of the final multiplexer from the base pattern $\mathbb{F}$ . . . 193
4.17 Base layout from which the `Sky130Mux` is composed . . . 195
4.18 Metal m_0 wiring atop the base `Sky130Mux` generator layout . . . 195
4.19 Idealised Sky130Mux circuit . . . 197
4.20 Actual Sky130Mux circuit . . . 198
4.21 Hierarchical 4:1 transmission-gate multiplexer layout for Global Foundries 180 nm . . . 199
4.22 Architectural overview and optional features produced by the CLB generator . . . 201
4.23 Layout strategy for the 4-LUT CLB . . . 203
4.24 CLB layout generated by `LutB` in its default configuration . . . 204
4.25 CLB circuit with default parameters . . . 205
4.26 4-LUT CLB generated with the option for a second bypass input . . . 206
4.27 Early 4-LUT layouts generated by `LutA` . . . 207
4.28 An early scaling pattern for k-LUT CLBs generated by `LutA` for $k = \{4, 5, 6\}$ . . . 208
4.29 Tuning `Sky130Mux` gate sizing . . . 211
4.30 Tuning CLB read-path buffer sizing . . . 212
4.31 Layouts produced by each of the four CLB generation flows . . . 216
4.32 Carry bit circuit . . . 218
4.33 Layout for a single bit of the carry chain, generated by `atoms::Sky130Carry1` . . . 220
4.34 A soft S-44 LUT daisy-chains two 4-LUTs . . . 221
4.36 Template for the `Sky130SimpleTransistor` layout . . . 224
4.37 Correspondence of two output `Sky130SimpleTransistor` circuit models to their input parameters . . . 227
4.38 Example layouts generated by the `Sky130TransmissionGate` class with select parameters . . . 229
4.39 `Sky130TransmissionGateStack` circuit for the sequence (A, B, C, D, E) . . . 230
4.40 Layout of a 5-terminal sequence of transmission gates, generated by `Sky130TransmissionGateStack` . . . 232
4.41 Transmission-gate-stack layouts with varying transistor sizes and net sequences . . . 233
4.42 N:1 Interconnect Multiplexer Circuit . . . 235

4.43 Layout recipe for the $N:1$ transmission-gate multiplexer 236
4.44 A $6:1$ one-hot transmission gate multiplexer generated by `atoms::Sky130Interconnect-Mux1` 240
4.45 $N:1$ multiplexers generates by `Sky130InterconnectMux1` 241
4.46 M:2 interconnect multiplexer circuit 242
4.47 $5:2$ Multiplexer layout generated by `Sky130InterconnectMux2` 244
4.48 $7:2$ Multiplexer layout generated by `Sky130InterconnectMux2` 245
4.49 Bundle layout strategies in the `InterconectWireBlock` 247
4.50 Skew between longest and shortest paths in `InterconnectWireBlock` layout regimes 247
4.51 An interconnect wire block laid out by `InterconnectWireBlock` 248
4.52 Preliminary machinery for laying out a tile 254
4.53 Layout of a UCA tile generated by `ReducedTile`, before automated routing . . . 255

List of Tables

1.1 Commonly used SkyWater 130 nm layers and their shorthands 23
1.2 Commonly used Global Foundries 180 nm layers and their shorthands 23

2.1 Distance functions between two points (x_0, y_0) and (x_1, y_1) 45
2.2 Supported intersections between shapes 47

3.1 UltraScale interconnect entity names and their assigned groups 114
3.2 The interconnect wiring in an UltraScale tile 118
3.3 Correspondence of non-local IIB Stage 1 nodes to regular IIB Stage 1 nodes . . 119
3.4 Synopsis of algorithmic steps when connecting interconnect wiring to IIB Stage 1 multiplexers 121
3.5 Final clean-up phase for north wires, $N^{(1)}$, $N^{(2)}$, after completing $\rho^{(N)}$ 122
3.6 Pre- and post-pattern parameters for $\rho^{(d)} : d \in \{N, E, S, W\}$ 122
3.7 Special connections from incoming interconnect wiring to IIB Stage 1 multiplexers 123
3.8 IIB Stage 1 multiplexer allocations to groups of like drivers 128
3.9 IIB Stage 2 to BLE connections, group C 137
3.10 IIB Stage 2 to BLE connections, group D 138
3.11 Correspondence of non-local OIB Stage 2 nodes to regular OIB Stage 2 nodes . 143
3.12 Synopsis of algorithmic steps when connecting interconnect wiring to OIB Stage 2 multiplexers 146
3.13 Final clean-up phase for south wires, $S^{(i)} : i \in 1, 2, 4, 12$, after completing $\lambda^{(S)}$. 147
3.14 Pre- and post-pattern parameters for $\rho^{(d)} : d \in \{N, E, S, W\}$ 147
3.15 Connections from interconnect to special OIB Stage 2 nodes 148
3.16 Position and quantity of non-local bounce nodes driving OIB Stage 1 157
3.17 Fan-out patterns between Stage 2 and Stage 1 of the OIB 158
3.18 Initial and final parameters for OIB Stage 2 to Stage 1 pattern instances 158
3.19 Interconnectivity of length-12 North, South, West wires in the interconnect crossbar 163
3.20 Connectivity from length-4 to length-12 wires in the interconnect crossbar . . . 164

4.1 Parameters exposed by the Sky130Buf simple buffer generator 173
4.2 Parameters exposed by the Sky130SplitBuffer split buffer generator 175
4.3 Parameters exposed by the Sky130Dfxtp flip-flop generator 176
4.4 Parameters exposed by the `Sky130Tap` tap cell generator 180

4.5 Parameters exposed by the `Sky130Decap` decap cell generator 183
4.6 Parameters exposed by the `Sky130Xor2` two-input exclusive-or generator 190
4.7 Standard-cell flows benchmarked against BFG 213
4.8 Performance of BFG CLB vs. synthesised CLBs 215
4.9 Comparison of UltraScale and UCA architecture parameters 253

Listings

2.1 Pin semantics, excerpt from `physical_properties_database.h` 29
2.2 Layer aliasing, excerpt from `dev_pdk_setup.cc` 30
2.3 Constraint structs, sanitised excerpts from `physical_properties_database.h` . 31
2.4 Documenting rules against confusion, from `dev_pdf_setup.cc` 34
2.5 Helpers for sizing typical via encapsulation by metal, excerpt from `physical_properties_database.h` 36
2.6 Circuit primitives database interface, from `circuit_primitives_database.h` . . 36
2.7 `Slice`s are light-weight references to `Signal`s, from `circuit/slice.h` and `circuit/wire.h` 38
2.8 Common transformations for each shape defined by `Manipulable`, from `geometry/manipulable.h` 41
2.9 Active layer stack 43
2.12 Using a saved anchor to guide grid alignment 43
2.13 Distance between rectangles, from `geometry/rectangle.cc` 46
2.14 Intersecting a line with an arc, from `geometry/arc.cc` 52
2.15 Casting rays to detect if a polygon contains or surrounds a rectangle, from `geometry/polygon.cc` 56
2.16 The `Line::Shifted` instance method will generate a shifted version of a line, from `geometry/line.cc` 63
2.17 An example of using the `MemoryBank` to simplify allocating memory instances to rows, based on excerpts from tiles/lut_b.cc 69
2.18 Snippet of the VLSIR `vlsir.raw.Layout` message 96
2.19 Snippet of the VLSIR `vlsir.raw.Instance` message 96
3.1 RapidWright program to extract interconnect nodes, courtesy of Jonathan Greene and Eddie Hung. © 2023 AMD; Apache 2.0 license. 106
3.2 Snippet of the (truncated) X20_Y100 interconnect edge list produced by RapidWright 110
4.1 Method to fill decaps into some span of a row, to the right, from `atoms/sky130_decap.cc` 183
4.2 The `StripIntoUnits` utility method, from `utlity.cc` 185
4.3 Creation of the SkyWater 130 nm to Global Foundries 180 nm layer aliases, from `dev_pdk_setup.cc` 199

4.4 Setting up a Global Foundries 180 nm cell instead of the SkyWater 130 nm one, from `main.cc` 200
4.5 Default configuration file for the `LutB` CLB generator 203
4.6 Verilog implementation of the default `LutB` CLB configuration 213
4.7 Structured access to layout properties post-parameterisation in `Sky130SimpleTransistor` 226
4.8 Size routing column width to fit overall cell with to pitch, from `atoms/sky130_interconnect_mux1.cc` 237
4.9 Method to import generated topology, from `tiles/reduced_slice.cc` 256
4.10 Invoking the router on the imported topology, from `tiles/reduced_slice.cc` . 256
5.1 Computing extension of polysilicon over diffusion in `atoms/sky130_simple_transistor.cc` 268
5.2 Comments for computing the extension of polysilicon over diffusion in `atoms/sky130_simple_transistor.cc` 269

List of Algorithms

1 Assemble IIB Stage 1 groups 126
2 Preparation and distribution of IIB S1 muxes over IIB S2 muxes 127
3 Generate IIB S2 Group A & B to BLE connections 139
4 Generate IIB S2 Group C to BLE connections 140
5 Generate IIB S2 Group D to BLE connections 141
6 Increment x by some minimum δ, skipping members of the special set $\sigma_j^{(O)}$. 147
7 Allocate BLE outputs to OIB Stage 2 muxes 153
8 Helper methods for allocation of BLE outputs to OIB Stage 2 muxes 155

Acronyms

AI Artificial Intelligence. 21, 272

AMS Analogue/Mixed-Signal. 1, 8, 15, 24, 169, 259, 272

ASIC Application-Specific Integrated Circuit. 2

BFG Berkeley Fabric Generator. 15

BLE Basic Logic Element. 10, 103

BSD Berkeley Software Distribution. 14

CAD Computer-Aided Design. 267

CI Continuous Integration. 260

CLB Configurable Logic Block. 10, 103, 167, 258

CMOS Complementary Metal-Oxide-Semiconductor. 4

CTS Clock Tree Synthesis. 266

DEF Design Exchange Format. 95, 169

DFT Design For Test. 262

DRC Design Rule Check. 4, 27, 94, 169

DSL Domain-Specific Language. 270

ECS Entity Component System. 267

EDA Electronic Design Automation. 2

FET Field-Effect Transistor. 4

FOSS Free and Open-Source Software. 16, 22, 169, 213, 260, 264

FPGA Field-Programmable Gate Array. 3

GCC GNU C Compiler. 14

GDSII Graphic Data System II. 95, 169

GPL GNU General Public License. 45

GPU Graphics Processing Unit. 3

GUI Graphical User Interface. 270

HDL Hardware Description Language. 6

IC Integrated Circuit. 4, 272

IDE Integrated Development Environment. 271

IIB Input Interconnect Block. 117

ILP Integer Linear Program. 21, 85, 264

ISA Instruction Set Architecture. 1, 13

LEF Library Exchange Format. 95, 169

LLM Large Language Model. 3, 22, 272

LUT Look-Up Table. 10, 103

LVS Layout vs. Schematic. 27, 94, 169

MIP Mixed-Integer Program. 21, 85, 264

ML Machine Learning. 272

MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor. 4

NDA Non-Disclosure Agreement. 260

OIB Output Interconnect Block. 140

OOP Object-Oriented Programming. 41

PAR Place-And-Route. 215

PDK Process Development Kit. 22, 27, 95

PDN Power Distribution Network. 266

PL Programming Language. 1

PLA Programmable Logic Array. 2

PLD Programmable Logic Device. 2

QP Quadratic Program. 85, 264

RISC Reduced Instruction Set Computer. 12

RL Reinforcement Learning. 21, 272

RPC Remote Procedure Call. 271

SAT Boolean Satisfiability. 21, 85, 264

SerDes Serialiser/Deserialise. 10

SPICE Simulation Program with Integrated Circuit Emphasis. 9, 27, 94, 169

SRAM Static Random-Access Memory. 10

STA Static Timing Analysis. 7

SWIG Simplified Wrapper and Interface Generator. 270

UCA Unnamed Columnar Architecture. vi, x, xii, 250, 252, 253, 255, 258, 271

VLIW Very Long Instruction Word. 2

VLSI Very Large-Scale Integration. 4

WSC Warehouse Scale Computer. 3

Chapter 1

Introduction

1.1 Some History

Solutions to modern computing problems are overwhelmingly implemented in software. System architects can employ software engineers who specialize in any of a number of tailored programming languages. Engineers rely on robust compiler to produce correct and fast programs. The compilers are the culmination of decades of research into programming language and compiler theory. The programs themselves are then executed by highly-optimized fixed-function processors on inexpensive commodity hardware platforms. If the engineers make a mistake they can modify the source code and try again within minutes at negligible additional cost.

General-purpose computing catalysed the digital revolution on room-sized mainframes ("programs") and on personal computers ("applications"), the internet ("web apps"), and mobile devices (just "apps"). This paradigm offers a cleanly delineated and compelling set of abstractions. Behind an Instruction Set Architecture (ISA), computer architects and circuit designers are free to optimize micro-architecture and hardware performance. Compiler designers focus on turning a human-readable description of the program into a maximally efficient deployment of these machine resources, while Programming Language (PL) researchers fuss over the usability and correctness of the languages themselves. Atop these systems we build multi-user operating systems, even more abstract hardware abstractions, the clean cocoon of user space, and globe-spanning computer networks.

As access to computing has grown, computer scientists and their colleagues have been empowered to solve problems across a much broader spectrum of human endeavour: biology, chemistry, physics, economics, the social sciences, and so on. Industrialists and entrepreneurs have been able to solve increasingly complex problems for businesses and consumers, creating trillions of dollars in economic value in the process. Much of this was due to the economics of computing itself. During the reign of Moore's law [68] (prior to 2010), both the performance of the underlying circuits and the volume of manufactured machines increased exponentially, while the cost of computation decreased in kind.

Two key drivers behind what was perceived as Moore's law were ongoing improvements in circuitry and manufacturing processes. Smaller transistors require lower voltage and less time to switch. Dennard observed [22] that the power density of ever-smaller transistors remained constant, meaning that ever-greater numbers of transistors could be packed into the same chip with the same power and heat budget. At the same time, the advent of standard-cell-based methodology and Electronic Design Automation (EDA) software rapidly expanded the number of transistors that could be manufactured on a single chip (what we called Very-Large-Scale Integration, VLSI). With more transistors at their disposal, computer architects could pursue micro-architectural innovations, such as branch predictors, that improved the performance of instruction execution. These architectural improvements compounded with circuit speedup. The upshot of this is that during this period, faced with a computational problem slightly outside the realm of feasibility, a reasonable strategy to solve it might justifiably have involved simply waiting six to twelve months.

Software flourished as an alternative to the design and manufacture of electronics purpose-built to perform the same tasks, the culmination of which was the Application-Specific Integrated Circuit (ASIC). Despite improvements in the chip design process, developing an ASIC was — and is — an expensive proposition. One must expect to sell high volumes of chips to justify the capital expense of developing, verifying and manufacturing a single one. With processors improving at rapid clip this made sense to do only for processors themselves and other high-volume, fixed-function devices such as audio, networking and graphics processors.

However, there existed some applications for which central processor performance was either a) perennially insufficient or b) overkill. For the former consider Digital Signal Processing (DSP), wherein computations usually decompose into linear algebra operations on vectors and matrices. These structures and the operations defined on them exhibit a high degree of parallelism which is not effectively exploited by a processor with a single thread of execution. This eventually motivated Very Long Instruction Word (VLIW) machines, vector machines and superscalar processors. As an example of where these are overkill, consider the controllers for traffic lights. These differ slightly between installations and are deployed in modest volume. They need to be simple, fast, reliable, and a little flexible; they do not need to be general-purpose computers.

Programmable Logic Devices (PLDs) have evolved developed to offer a half-way measure between software running on fixed-function processors and the difficulty, time and cost of developing ASICs. Although they have come in a variety of types, PLDs all work by providing a large (and fixed) set of logic resources that can be enabled, disabled, or rerouted after manufacture. For example, Programmable Logic Arrays (PLAs) provide two consecutive logic regions wherein gates can be individually disabled by built-in fuses. The output of each region can be any combination of its inputs. The first plane is configured to implement "AND" logic, and the second to implement "OR", so that each output can implement any boolean function of its inputs in disjunctive normal form. These devices enjoyed modest adoption among various niche use cases.

1.2 A New Golden Age of Computer Architecture

Neither Moore's law nor Dennard's scaling law are still in effect [102]. The rapid pace of processor performance improvement described in Section 1.1 has slowed to a respectable, but no longer spectacular, 15% per year. Transistor density is also improving at ∼15% per year, compared to ∼50-60% previously [64], [119]. We are reaching the physical limits of the circuit technologies we have relied on since their inception.

Computer architects have instead been forced to turn their attention to exploiting any available parallelism and scaling the sheer number of cores, processors, and computers in existence. Large corporations — so-called "hyperscalers" — have built Warehouse Scale Computers (WSCs) aggregating tens to hundreds of thousands of processors. We have had to smarten how we distribute workloads: older machines are still perfectly suitable for some tasks and therefore remain economical to run. Some hyperscalers today offer machines running processors from up to 11 generations ago[1].

We must revisit the general-purpose computing paradigm. Some computational loads are now important and commonplace enough that the next leap in their performance can only come from specialised ASICs. The opportunity this presents for the design and innovation of such accelerators and other application-specific architectures has led Hennessy and Patterson to declare a new golden age of computer architecture [41]:

> The next decade will see a Cambrian explosion of novel computer architectures, meaning exciting times for computer architects in academia and in industry.

In the new regime, domain-specific accelerators will become crucial parts of systems that demand ever greater performance. Consider the Graphics Processing Unit (GPU), which was initially invented to accelerate graphics processing. Computing an image for display can usually be reduced to many parallel operations to compute the state of each on-screen pixel. GPUs are therefore built as large vector machines with fast access to shared memory. This architecture also happens to be amenable to some classes of physical simulation and, most recently, general matrix multiplication for the large linear-algebra workloads in various fields artificial intelligence. Indeed, recent breakthroughs in the capabilities of Large Language Models (LLMs) can be attributed to capacity of the underlying compute to train on large volumes of data.

PLDs do not directly compete with domain-specific accelerators that can economically justify implementation in an ASIC. Field-Programmable Gate Arrays (FPGAs) in particular, however, should be of greatly renewed interest to architects and system designers who are experimenting and evaluating new architectures, or who are in search of performance where the volume of devices is low. FPGAs, too, should enjoy a new golden age. Before we can visit FPGAs, however, we must review the broader field of Electronic Design Automation.

[1] At the time of writing, Google — a hyperscaler — currently offers [35] machines with processors released from 2011 up to and including 2026.

1.3 Electronic Design Automation

We refer to the complex electronic circuitry that is packaged into a single physical chip as the Very Large-Scale Integration (VLSI) of Integrated Circuits (ICs). This is holdover from the graduation of hand-crafted ICs, in the late 1960s, to significantly more complicated devices that required computational help to build. Even though the complexity of such circuits has increased by orders of magnitude since, we still refer to them as Very Large. After several decades of iteration, the development of VLSI ICs has been bifurcated into two remarkably distinct disciplines, each with their own sets of related but distinct tools, best-practices, taboos, mantras, and required texts. They are 1) the design of digital circuits and 2) the design of analog/mixed-signal circuits, which are everything else. In order to understand how any new tools might fit into this landscape, we must quickly review the basics.

1.3.1 CMOS

Today, the most popular fabrication technology for ICs is known as Complementary Metal-Oxide-Semiconductor, or CMOS. The "metal-oxide-semiconductor" (MOS) part of the name refers to physical arrangement of materials needed to form a Field-Effect Transistor (FET): at the base sits a semiconductor (such as silicon), above which sits an oxide to act as a dielectric, atop which sits a polysilicon contact, forming a *gate*. The semiconductor is doped with positive or negative charge carriers which collect near the dielectric boundary when a relative voltage is applied to the metal terminal. When collected, the charge carriers form a conductive channel and the field-effect transistor is turned on, because of the *effect* of the electric *field* at that point. The technology is "complementary" because transistors made of both positive (PMOS) and negative (NMOS) charge carriers can be manufactured together, and therefore participate in the same circuit.

To manufacture a CMOS device we use *photolithography* to mask-off (like a painter would a house) and then deposit successive layers of dopants, metals, and other materials. A *mask*, or pattern, is used with a light source to guide where each physical structure lands. A thorough description of CMOS technology and the theory of operation behind MOSFETs can be found in [78]. For the purpose of this discussion, note that MOSFETs have physical dimensions that form design-time considerations: the area of the gate, for example, determines the width of the conducting channel and therefore the amount of current that it can support. The wires that are used to connect transistors must be created by a separate photo-lithographic process that guides metal deposition to the right places. Each layer of material, and so each layer of wiring, requires its own lithographic process.

The light source, the physical construction of the masks and the limits of precision within each tool necessitate a series of rules to be followed when making each pattern. These Design Rule Checks (DRCs) include limits on the minimum area of shapes; the minimum and maximum widths of shapes; minimum and maximum spacings between shapes; the minimum density of shapes; and so on. Each layer has its own sets of constraints. Some layers are constrained with respect to one another. Our goal as IC designers, and the fundamental

problem of physical design in VLSI, is to send the manufacturing *foundry* the set of masks — one per layer — that they can use to manufacture our circuit, all the while adhering to the foundry's rules in order to a) have some confidence that the device will work and b) not waste our money. We call this set of patterns the design's *layout*.

1.3.2 Digital Design

Digital circuits can be reasoned about as compositions of boolean functions, meaning that the value of any signal can take only one of two values, say "0" and "1". The functions themselves are made up of familiar boolean logic operations which we call *logic gates*:

- "AND", $Y = A \wedge B$, also written $Y = AB$;
- "OR", $Y = A \vee B$, also written $Y = A + B$; and
- "NOT", $Y = \neg A$, also written $Y = \overline{A}$.

In *combinational* digital circuits the signals propagate simultaneously through all logic branches, and in *sequential* circuits signals are latched — or stored — by a special signal we call the *clock*. Clock ticks (cycles) drive logic through the sequences of states that will ultimately lead to some computed result or action. The ticks themselves dictate when signals must first arrive at storage elements (setup time); how long they must be kept constant there (hold time); and how soon they are available for use thereafter (propagation delay). Therefore, the correctness of a digital circuit is principally a function of 1) the soundness of the underlying composition of logic gates and 2) the timing of signals travelling through and between them.

In fact, we can completely describe the digital abstraction by adding a few more states, tracking whether a signal is in an unknown state ("U"), unimportant state ("X"), or mysterious "other" state such as high-impedance or disconnection ("Z"). In reality, each of these represents some a range of underlying voltages or circuit conditions that may or may not be anomalous, but the details simply don't matter.

Digital circuits thus enjoy a relatively simple set of abstractions, which in turn allows room for complexity within designs themselves. With computer-aided automation, we are able to specify, design and verify vastly complicated digital circuits: superscalar out-of-order processors, massively parallel vector engines, and so on. Today's designs contain hundreds of millions to billions of logic gates, far more than can be reasoned about by a human. We are able to *verify* that these circuits behave as intended because it is feasible simulate the circuit tick-by-tick: since each signal has a small state space {0, 1, X, U, Z}, we can accommodate a large number of signals and a large of time steps. In order for these abstractions to hold, a great deal of effort must be spent to keep each of the physical circuits within their operating limits, and to keep the timing of signals within allowed margins.

A significant body of work is devoted to the automatic generation of digital circuits. The problem is typically partitioned into successive steps that, together, are known as the VLSI

flow. These steps take the design from a high-level description of functional requirements all the way to a manufacturable blueprint, and are broadly: 1) system specification; 2) architectural design; 3) functional and logic design; 4) circuit design; 5) physical design; and 6) physical verification and sign-off. We have just reviewed step (3), and in what follows we will briefly review steps (4) to (6). For a complete and contemporary treatment of the field, the reader is referred to [48].

Digital circuit design requires taking high-level descriptions of combinational and sequential boolean functions, written in Harware Description Languages (HDLs) like Verilog or VHDL, and mapping them to directed graphs of the basic logic gates (AND, OR, NOT). This process is known as *synthesis*. These logic gates are then mapped to libraries of premade circuits known as *standard cells*, that implement each function using conventional electronic circuitry. This process is known as *technology mapping*.

Standard cells were a revolutionary abstraction for digital circuit designers. Instead of having to design and test bespoke circuits (and their blueprints) for each new boolean function, standard cells enabled large functions to be decomposed into units of proven circuits that are made elsewhere. This division of labour in turn enabled experts to optimise the implementation of each cell, and elsewhere the complexity of large digital designs, completely separately. Standard cells are packaged in *libraries* for a given semiconductor manufacturing process from a given manufacturer. Libraries contain cells for a variety of logic functions, starting with the basics above, but extending to common functions such as:

- "XOR", exclusive-or, $Y = A\overline{B} + \overline{A}B$, also written $Y = A \oplus B$;
- "NAND", not-and, $Y = \overline{AB}$;
- "AOI", and-or-invert, $Y = \overline{AB + C}$;
- "OAI", or-and-invert, $Y = \overline{A(B + C)}$
- "MA", the majority function, $Y = AB + BC + AC$;
- buffers;
- latches and flip-flops; and
- multiplexers.

Standard cells greatly simplify the problem of automatic circuit generation. The cells in a particular library are designed to fit into rows of regular heights and with regular widths, and can either be rotated 180° or not at all. Before standard cells, designers would have to reckon with the placement of all of the *transistors* in their designs, still constrained by a manufacturer's design rules but nonetheless contending with a much larger space of possibilities. Because standard cell placement is discretised into a fairly limited number of fixed *sites*, the problem is much simpler.

A closely related problem to placement is that of *routing*, the process of wiring logic devices together automatically. Routers must generate sets of wires that simultaneously connect all of the necessary parts of a circuit without violating any design rules: no short circuits, no wires too thick, no wires too thin, no wires too close too each other, and so on. Although they can simultaneously optimise for competing objectives, routers are typically tasked with minimising the delay of the slowest signal path within a clock cycle (the *critical path*), thereby maximising the frequency at which the clock can be run. To achieve this, routers employ another convenient abstraction: a *routing grid*. Routing grids limit wire placement to a rectilinear set of tracks, set apart at a regular pitch, with metal layers typically dedicated to alternatively horizontal and vertical wiring. Wires can only appear on a track and can only change direction at the intersection between tracks, typically by changing layers through a via. This set up allows a router to find paths without the need for whole classes of invariant checks in its core loop, thereby greatly simplifying the problem.

Placement and routing are coupled problems: placement must consider the wiring needed to connect devices, and routing must consider where all devices have been placed. Standard cells ameliorate the routing problem in two major ways. First, they encapsulate the wiring for a complete function internally, exposing only *pins* for connection to external wires. This reduces the total number of wires that the router must simultaneously solve for, and therefore the overall problem size. Second, pins are positioned such that, when the cell is situated legally, they fall onto a predefined routing grid. *This* makes it easier for the router to connect wires to the cells, reducing the search space. Again, the complexity of the problem is reduced by an abstraction that discretises the solution space coarsely.

Unfortunately, just as there are no free lunches, there are costs to these abstractions. Subsequent steps in a typical digital design flow aim to manage these costs through *design verification*. Before individual circuits (such as those in the standard cells) can be used, they are *characterised* through analog/mixed-signal simulations (Sec. 1.3.3) that try to bound their behaviour within a larger circuit. This is done once a circuit is finished and need not be repeated every time it is used. Characterisation determines physical properties at port interfaces, such as the input capacitance of each port, but also the determines the timing behaviour of signals between ports. Best- and worst-case delays are found for all logically-coupled ports under stated assumptions about the input signal shape and the output load. Static Timing Analysis (STA) tools use can use this data to reason about whether signals in each logic path will arrive within the required timing windows. Having fixed cells with pre-existing characterisation data speeds STA up considerably, but the process is fundamentally conservative. Since STA tools propagate the upper and lower *bounds* of signal timing through a circuit, they trade an accurate representation of the final circuit for correctness and speed.

Following — and perhaps as a result of – STA, various steps are taken to patch problems with circuit timing. We shall call these "mitigations" because they minimise the consequences of abstractions earlier in the VLSI flow. A first such mitigation is *retiming*, as described in [65]. Combinational logic is moved through synchronous boundaries (that is, registers) to re-balance the combinational signal path before and after clock edge, taking care to ensure that all downstream logic remains correct.

A second such mitigation is *buffer insertion*. A typical standard cell library contains a few hundred cells, mostly made up of the basis functions from listed above (XOR, NAND, AOI, etc.) but with variations on the number of inputs and on the output *drive strength*s. Drive strength is a measure of how much load a cell can drive: synthesis tools must guess, based on the number of downstream cells, which variation of cell should be used. This dictates the size of the cell and, therefore, where the placer can put it. However, the wires that connect the cells are not perfect conductors, and the length of those wires has a significant impact on what drive strength is needed (see Ch. 4 in [78]). Yet, it isn't until after routing is completed that we can evaluate how long the wires are, and therefore how strong the original cell needed to be. Buffer insertion aims to correct for cell sizing choices and to compensate for wiring delay by inserting buffers that rejuvenate signals inline.

A third such mitigation is *clock tree synthesis*. We've said that the timing of all signals within a digital circuit depends on the clock, but because the clock must be distributed by physical wires the clock itself experiences propagation delay and signal degradation. Worse still, since the paths from a clock source to any given part of the circuit differ, different regions of the circuit will receive the clock offset with respect to one another. This is known as clock *skew*, and it is best mitigated by fractal wiring structures such as H-trees [13] that deliberately minimise such variations.

Placement and routing are both NP-hard problems at the core of the physical design problem, and remain the subject of extensive research and development. Modern algorithms are very effective at correctly placing and routing millions of cells with complex constraints. The abstractions we have discussed have played a critical role in making these problems feasible *today* for the design of *tomorrow*'s chips. However, we have also seen that they come with notable costs.

1.3.3 Analog/Mixed-Signal Design

In reality, electronic circuits live in the physical world and are subject to environmental temperature and pressure, noise, cosmic radiation, manufacturing defects, physical shock, and so on. There is a rich field of work designing circuits that not only cope with these analogue realities, but that can even exploit them. Consider radio equipment, audio amplifiers, and sensors as just a few examples. Digital circuits are, fundamentally, analogue circuits with bistable behaviour that we can abstractly treat as digital. Even though it is fundamental, we distinguish the design of Analogue/Mixed-Signal[2] (AMS) as a separate discipline to digital design because of the vastly different tool-sets, both theoretical and computer-aided, that are involved. Razavi summarises it [79]:

> Good analog design requires intuition, rigor, and creativity. As analog designers, we must wear our engineer's hat for a quick and intuitive understanding of a large circuit, our mathematician's hat for quantifying subtle, yet important effects in a circuit, and our artist's hat for inventing new circuit topologies.

[2] "Mixed-Signal" means that at least some part of the design is analogue.

Unlike the digital design flow, AMS circuit designers enjoy, or suffer, from a significantly larger design space. Concretely, devices such as transistors, resistors, capacitors and inductors are sized and placed individually, usually from a large range of possible dimensions. Components are placed in two-dimensional space with a fine-grained, if not effectively continuous, coordinate system.

The need for deft craftsmanship applies both to the circuit's abstract design *and its physical layout*. The design of circuits themselves remains a rich field of research, while the layouts are typically performed manually by skilled practitioners. Still, the two are tightly coupled. In designing circuits for VLSI applications, AMS designers must contend with minute details of how transistor behaviour varies with environmental and manufacturing factors. The design rules we introduced in Section 1.3.1 are augmented with best-practices for layout to mitigate, for example, the variation that can occur between even neighbouring transistors. In return for this labour, AMS designers are singularly placed to push the limits of our technologies. Even within a digital system, the high-speed internal serial links, multi-gigabit-per-second wireless networking, and high-colour high-framerate display technologies were first made possible by hand-crafted and finely-tuned mixed-signal ICs.

The necessary automation to handle this complexity starts, and sometimes ends, with verification. The most general, and therefore most useful, way to predict the behaviour of *any* electronic circuit requires us to computationally solve for the system's behaviour as time evolves, which usually means expressing the circuit as a system of equations and solving for currents and voltages using Kirchoff's laws. Abstractions, where possible, serve to simplify the physical behaviour of each component. Non-linear devices (such as transistors) are represented by "compact models", and these models are queried to determine how the devices respond to changes in conditions at their inputs and outputs. Whether from closed analytical solutions (such as for capacitors and inductors) or from such compact models, device behaviour is first made differentiable with respect to time. Simulation then proceeds in (small) time steps, with changes in state from the last step being applied to the next, until the simulation is stopped.

The standard software system for such simulations is SPICE[3]. SPICE simulators are a broad class of circuit simulators that trace their origins to the first SPICE program [69]. The compact models that describe transistors, such as BSIM [90], are themselves the subject of significant research. Though they are imperfect, the resolution and generality of SPICE simulatiors make them the gold standard predicting the behaviour of AMS circuits.

In order for SPICE simulation to accurately reflect circuit behaviour, the circuit itself must be an accurate reflection of the physical device. At the micrometre and nanometre scales of modern CMOS manufacturing processes, significant electrical effects arise from the unintentional coupling of various materials in the process stack. These are known as *parasitics*. Parasitic capacitances appear where charge carrying materials couple to each other through dielectric materials. Parasitic inductances are likewise an emergent and unavoidable property of the physical structure. Parasitic resistances capture the internal, geometry-dependent

[3]SPICE is an acronym for Simulation Program with Integrated Circuit Emphasis.

resistances of process materials. The process of *parasitic extraction* aims to predict the values of these parasitic circuit elements from the circuit's intended layout based on models provided by the semiconductor foundry. These values are then used to augment the designer's circuit for simulation in order to better predict circuit behaviour once it has been manufactured. Whereas digital flows, which also undertake this step, try to avoid making it the digital designer's concern, AMS designers must consider it as a matter of primacy.

The vastness of the AMS design space, including unintentional environment and physical effects, and the necessarily fine granularity of simulations make verification of very large circuits impractical. Not impossible, but slow, taking up to on the order of days. Since it is often impractical to apply these techniques to large-scale digital systems, the appeal of digital design (Sec. 1.3.2) to a computer architect is at-once clear.

Finally, it should by now come as no surprise that digital circuit design operates within a convenient fantasy atop an analog world. Digital chips must all interface with this reality eventually and, as a result, digital tools all include mechanisms to include AMS blocks wholesale in their designs. These blocks, called *macros*, provide the physical implementations for variously complex functions that need only be marked as abstract (or "black box"). Such extensibility enables designers to assemble a chip from the works of many, with specialised and even out-sourced implementations of entire design components. Macros can be analog, digital or mixed-signal; typical examples include memory arrays (such as SRAM), well-optimised functional units such as multipliers and adders, passive components such as inductors and high-speed Serialisers/Deserialisers (SerDes) for input/output interfaces.

1.4 Field Programmable Gate Arrays

FPGAs are novel among PLDs because of how they implement digital logic (Sec. 1.3.2). Instead of arrays of AND and OR gates, fuses, and/or non-volatile memory, FPGAs are made up of many SRAM-based Look-Up Tables (LUTs). Static Random-Access Memory (SRAM) is typically made from bistable CMOS structures such as latches and flip-flops; it is fast and volatile, meaning it must be configured after each power cycle but also that it can be configured quickly. A k-LUT is a 1-bit-wide memory 2^k entries deep: it maps a k-bit inputs to either 0 or 1. As such, LUTs can implement *any* boolean function of k arguments. FPGAs work by decomposing any digital logic circuit into functions of at most k-many inputs, loading these functions into the LUTs, and connecting the LUTs over a programmable interconnect. Compared to more traditional PLDs, engineers choose FPGAsfor their greater logic capacity, reconfigurability, and interconnect bandwidth [108].

In order to support synchronous designs, LUT outputs are paired together with a register that is clocked by a distributed global signal. This structure is called a Configurable Logic Block (CLB) or Basic Logic Element (BLE) (terms we will use interchangeably). In traditional academic architectures, these CLBs are placed regularly through the fabric in between horizontal and vertical channels of interconnect wiring. As illustrated in Figure 1.1, the result is known as an "island-style" array, so-named because the CLBs look like islands

in a sea of interconnect wiring. In later chapters, we will show how BFG can create the circuitry for these components, as well as alternative interconnect architecture altogether.

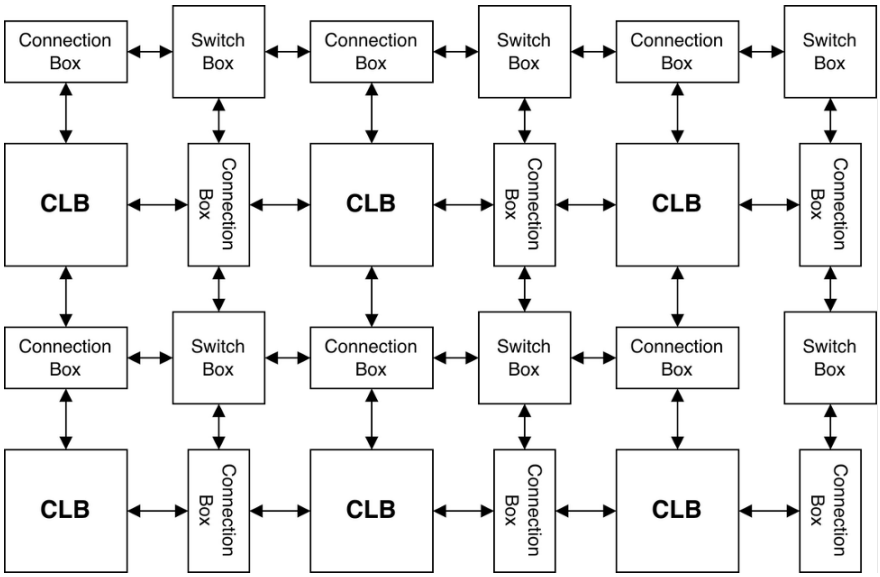


Figure 1.1: Traditional island-style FPGA architectures

The design of the interconnect networks is the subject of a rich field of research. To summarise, CLBs connect to vertical and horizontal wire channels through programmable crossbars known as Connection Boxes. At the intersections between horizontal and vertical channels another programmable crossbar connects wires together, allowing signals to change direction. The uni- or bidirectionality of the wiring in each channel, the composition of wire lengths, the exact pattern of connectivity in each crossbar (and to what extent they are depopulated), and the types of programmable switches employed are all common design considerations. An in-depth treatment of these topics can be found in [61].

To be used effectively, FPGAs require a complex toolchain to synthesise, map, place and route a high level logic description using the fixed resources of a particular FPGA device. This process is similar to the digital design flow for ASICs (Sec. 1.3.2), and identical at the front end[4]. Users specify designs in the same RTL, but the synthesis problem now requires packing logic into fixed-size LUTs instead of mapping them to fixed-function cells. The techniques are similar [66] and just as fascinating; the reader's curiosity is encouraged. Elsewhere, the placement problem now concerns existing LUT sites and routing is limited to fixed-position but very limited wiring.

[4]The similarities between the FPGA and digital ASIC flow contribute to the popularity of FPGAs for ASIC emulation.

1.5 Motivation

Like much of human endeavour, this work was initially precipitated by a profound sense of frustration — in this case, with the second-class role of FPGAs in the computing landscape. FPGAs have promised (Sec. 1.4) a fundamental shift in the computing paradigm since their inception in the 1980s. Their runtime reconfigurability, modest per-unit cost, low latency and high throughput were heralded as a boon for the computer architect's toolbox. Innovations in their circuitry and in their programming models were supposed to make FPGAs a better choice for a greater many classes of computation. In spite of all of this promise, FPGAs remain a niche solution to a great many niche problems.

The reasons for this, as we see it, are threefold. First, for the starting two decades of their lives FPGAs competed poorly with the rapid improvements in general processor performance afforded by Moore's Law. If one needed more compute, it was generally easier to wait 12 months for a 100% faster CPU *and simply recompile one's existing code* than it was to figure out how to get an FPGA to do the work. Where low-latency and/or high-throughput workloads existed, with the usual caveat that they did not also justify the development of an ASIC, a niche for FPGAs developed.

Second, once commercial FPGA vendors found success serving these niches, they started to optimise their product lines *for* these niches. In the last two decades, FPGA offerings have developed into Swiss-Army knives of compute: a basic fabric permeated by accelerators: digital signal processing processors, multiply-accumulate units, matrix multipliers, network interfaces, memory interfaces, complete RISC CPU cores, etc. They are pretty good at a great many things, but never the best. In closing remarks at a leading FPGA conference in 2019, your author witnessed a prominent member of the FPGA research community deliver a scathing rebuke of manufacturers' bent toward complexity. "Just give us a basic fabric," they probably said.

Third, because they did not find general use, FPGA devices became increasingly specialised for various niches. This fractured any nascent ecosystem before it was formed. FPGAs continued to be developed largely independently, in various commercial silos, by different teams and for different customers. Each FPGA had slightly (to vastly) different capabilities, architecture, and toolchains. Vendors would not share the details of their designs and so any new designer — as well as anyone in academia — was forced to guess and then invent something else. The result is that there is no consistent set of capabilities and no consistent programming model for FPGAs. Before one can use such a device, one must become an expert in the target platform, understanding its architecture, capabilities, and programming quirks.

To further illustrate the barrier to FPGA adoption, consider the flowchart of Figure 1.2, which documents the typical (if anecdotal) experience of someone looking to start computing with FPGAs. In addition to navigating the myriad different devices available, any would-be FPGA engineer must first contend with buying and setting up a development platform and its corresponding development tools. These are proprietary, platform-specific and restricted by onerous licenses. The tools, which must perform similar synthesis, placement and routing

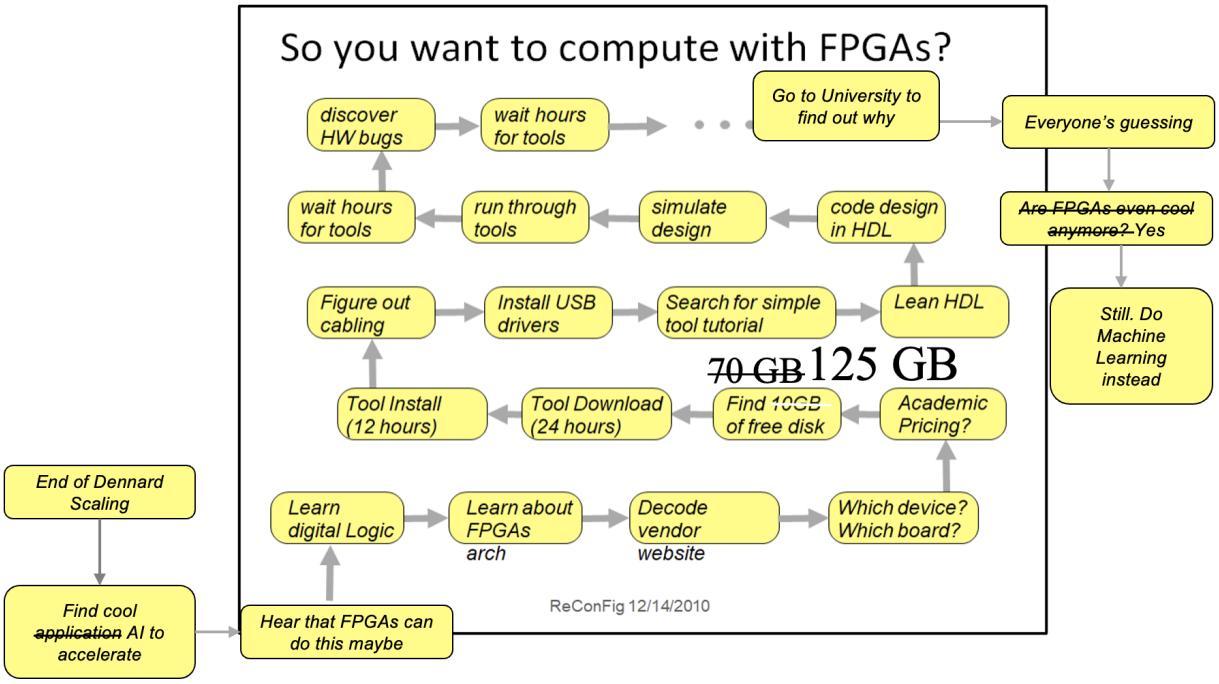


Figure 1.2: So you want to compute with FPGAs?

tasks to those discussed in Section 1.3.2, are also very slow. A single run can take up more than a day for a moderately large design. On one hand, this may seem like a similar burden to that faced by IC designers, who dutifully accept their fate and silently suffer a similar design cycle with a similarly niche set of tools (more on that in Chapter 5). On the other hand, the economics of ASICs make them largely insensitive to the pain of tooling. We believe that for FPGAs, which have long had competition in terms of computing paradigm, cumbersome tooling is a meaningful disadvantage.

When we compare the FPGA engineer's experience to that of a software engineer, the differences are stark. Software engineers can choose from any number of languages that target any number of portable virtual or physical machines with abstract ISAs. They can choose from different compilers to compile their code, different virtual machines to run their code, different vendors for equivalent processors, memory, motherboards, and so on. Compilation can be incremental and fast, shortening the design cycle to *minutes* over hours (something partial reconfiguration for FPGAs attempted to achieve). Software engineers benefit from the distillation of decades of ideas among competing groups into robust, portable, and well-supported tools. These tools run on ubiquitous hardware that is readily commoditised through a multitude of industry standards.

However, we note that general purpose computing entering the 1990s suffered fragmentation similar to the current FPGA market. Back then, there were many more vendor-specific

devices, each with their own operating systems and quirks, and many more competing architectures, each with their own vendor-specific compilers. This should sound familiar. Though the fragmentation was ultimately resolved by a variety of commercial forces, an outsize role can be attributed to the then-nascent open-source software movement. Within the span of the next two decades, various independent efforts into like tooling, such as into compilers and operating systems, coalesced into fewer, larger, volunteer-driven (and eventually commercially funded) open-source projects. Prominent examples are the GNU C Compiler (GCC), which grew to replace the once-indomitable Intel C Compiler; the Linux kernel and the various Linux distributions, which grew to replace Unix (and even Microsoft Windows servers); and the Berkeley Software Distribution (BSD), a Unix alternative, that led the first wave of internet platforms and that eventually led to Apple's macOS.

We believe a significant factor in the success of open-source software is that it enables ideas to be shared, reviewed, iterated upon and hardened, at rapid pace. Many competing ideas can start from the same point, and as the merits of each become clearer with time, winners emerge. In contrast to the rapid enablement of software ecosystems, FPGA development has not meaningfully changed in the last two decades. The first version of Figure 1.2 shown by Prof. John Wawrzynek in 2010 [109], while the version you see above was updated for 2025. Not much changed in those 15 years, save for the order of magnitude increase in the size of the required tool download. The experience has remained largely the same.

Therefore, a central thesis of this work is that enabling the community-driven development of accessible and *real* FPGAs will stimulate significant innovation in FPGA performance, usability, and adoption. As we will see in the next section, others have also recognised this and have already undertaken several efforts to develop open FPGA architectures and, separately, tooling. We believe that, not only is extending these efforts to the underlying circuits and physical design necessary to bridge the gap with commercial devices, but it is already readily achievable. Therefore, we believe that the community would do well to focus its efforts into building co-optimised circuits and layouts for FPGA fabrics.

To that end, we seek to show that programmatically generating physical designs for FPGAs yields significant performance benefits over more general approaches and, importantly, that it is feasible. In contrast to existing physical design automation (Sec. 1.3), which tackles more complex and more general problems of placement and routing, BFG aims to show that concerted effort into a choice but limited set of cells can yield significant benefits over more general-purpose approaches. First among the benefits we foresee is the improved performance of the fabrics themselves. Second, codifying circuit and layout best-practice in response to various architectural choices will document those ideas together with the means for others to improve upon them. Third, if nothing else, the existence of such a repository will enable the training of future designers, both human and artificial.

1.6 Aims and Hypothesis

The overarching aim of this work is to build a parameterised generator for high-performance FPGAs. We seek to close the gap between public-domain and commercial devices. To that end, we aim to build an FPGA generator that prioritises circuit performance by programmatically generating custom circuitry and IC layout. It will, in-effect, be a silicon compiler for the "bottom-up" assembly of reconfigurable fabrics. The tool, which we shall call the Berkeley Fabric Generator (BFG)[5], should eventually:

W1. produce high-performance FPGA fabrics with a variety of architectural features;

W2. allow users to configure the architectural features at runtime, features such as:

a) the LUT size k;

b) whether LUTs are fracturable;

c) the type of configuration memory;

d) the configuration frame width; and

e) the mix of hardened arithmetic blocks, processors, and the like;

W3. produce an architectural description that allows external tools to map designs to and generate a configuration bitstream for the fabric; and

W4. be proven by tape-out with real silicon.

By *high-performance* we mean that the generated designs must offer compelling performance trade-offs in the design space, with features and performance that are relevant to modern users, and that they must be manufacturable. Figure 1.3 summarises this view.

We contend that the design space for FPGAs, while large, is not so large that the general-purpose standard-cell digital flow (Sec. 1.3.2) is necessary. On the contrary, the design-space is limited enough that it is already feasible to develop custom circuit and layout generators for most common features. Our hypothesis is that such a generator, focused on physical design, will net sufficient performance benefits over standard-cell-based generators to justify its inevitable complexity.

The requirements W are our wish-list items. The more specific aims of this work are to test our hypothesis while equipping BFG with the prerequisites to eventually build whole FPGAs. That is, BFG must first:

R1. be capable of fully expressing any AMS design;

R2. allow users to specify circuit parameters at runtime, such aspect ratio, transistor sizes, port selection, and so on;

[5]This is the best I could come up with after 7 years. You can also call it the Big F— Generator, as I do sometimes.

R3. produce common collateral for EDA tooling so that its output designs can be imported into existing flows as macros;

R4. produce designs that are DRC-correct by construction; and

R5. support design verification through established external tools over industry-standard interfaces.

Additionally, BFG should:

R6. be simple to use for those interested in re-using existing work;

R7. enable, or at least not prevent, the process portability of designs; and

R8. be completely Free and Open-Source Software (FOSS), and have only FOSS dependencies.

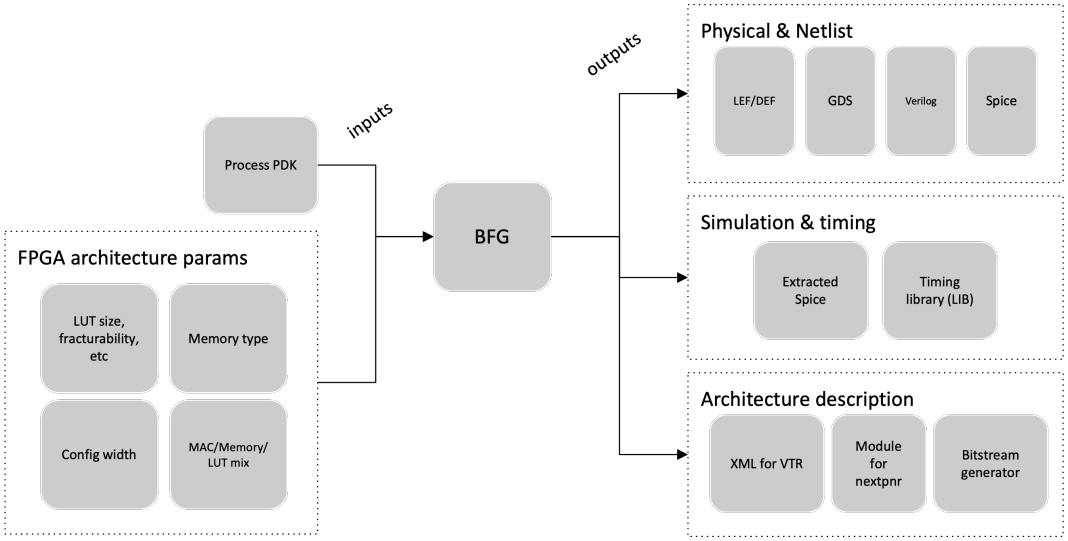


Figure 1.3: Desired BFG inputs and outputs

An additional and underlying aim of this work is to produce central repository for circuit and layout design ideas — together with the recipes for their generation in software — that research community can contribute to. We hope that:

- great ideas like design relocation, rapid reconfiguration and dynamic power scaling can play a bigger role in future fabrics if they can be offered to one another by researchers;
- the barrier to contribution is reduced by emphasising simplicity and providing starting examples; and
- we can re-invigorate innovation in reconfigurable computing by giving FPGA architects the means to physically realise their ideas, perhaps as useful devices.

1.7 Related Work

We restrict our survey to open source work academic or otherwise, not least because it is difficult to confirm even our staunchest suspicions about anything else. There are broadly three categories of related work to consider. First, existing community efforts into open-source tooling for FPGA fabrics (Sec. 1.7.1) have overcome a major barrier in the adoption of open-source fabrics, leaving us free to focus on physical design. Second, although BFG introduces an AMS framework of its own (Ch. 2), this was only born of necessity. There are now several alternate frameworks with differing approaches to common problems (Sec. 1.7.2). Third, FPGA generation is both old and new. We consider past approaches to automatically making realisable FPGAs as well as the recent projects that share much of our motivation (Sec. 1.7.3).

1.7.1 Open-Source Toolchains

In recent years, the independent efforts of many EDA researchers have coalesced into well-supported open-source packages that can perform end-to-end synthesis, placement and routing onto FPGA fabrics. VTR [26] is an established open-source tool for just that: it supports a flexible FPGA architecture description, a suite of architectural components, and has long been used by researchers to explore architectural and algorithmic ideas. More recently, Yosys and nextpnr have emerged as performant tools with an emphasis on targetting production FPGAs.

A notable feature of nextpnr is its emphasis on targetting production FPGA devices. Since manufacturers do not reveal the details of their designs, a considerable amount of reverse-engineering is required to understand a device's architecture and programming. There are over a dozen separate community projects each aiming to deconstruct a different commercial platform, including the Lattice Semiconductor iCE40 [112], ECP5 [88], and Nexus [113]; the Gowin Semiconductor LittleBee and Aurora V [97]; and the Xilinx 7-series [103]. Separately, the Bitfiltrator project [50] attempts to understand the programming *bitstream* (file) for Xilinx UltraScale devices. In doing so the authors are able to extract some architectural parameters, but not a complete picture of the architecture's circuit- and physical-design.

The progress made by these tools is what encourages us to pursue circuit and physical design. We showed in [8] that the performance of the open-source ABC synthesis tool [11] was nearing that of commercial state-of-the-art synthesis tools. In recent years, the CHIPS Alliance has started to add support for FPGA tooling projects. Their projects include the F4FPGA project [19] [103], which is an amalgamating end-to-end toolchain that calls itself the "GCC of FPGAs"; and the FPGA Interchange Format (FPGAIF) [20], an intermediate representation for FPGA architecture.

Open-source tooling for ASIC development has also matured. The OpenROAD project [1] combines various open-source tools for the end-to-end implementation of a standard-cell digital flow (Sec. 1.3.2). Its notable components include RePLace [17] for placement,

TritonRoute [47] for routing, TritonCTS [39] for clock tree synthesis, and OpenSTA [75] for static timing analysis. Combined with open-source CMOS processes (Sec. 1.7.5), researchers, students, enthusiasts and entrepreneurs finally have access to chip-building software that is complete free and open-source. The contributors to these efforts have paved the way for tools like BFG.

1.7.2 Open-Source AMS Frameworks

The venerable Magic VLSI [72] remains a cromulent, open-source method of programmatically generating VLSI layout. Its disadvantages are runtime performance, insistence on programming through Tcl/Tk, and a lack of interoperability with modern libraries. KLayout, a tool for viewing layouts, also includes scripting abilities in both Ruby and Python. These features were nascent while BFG was being developed, though it is possible that they could now be used to implement BFG's cell generators. Both Magic and KLayout principally serve as layout viewers and DRC checkers.

The Berkeley Analog Generator (BAG) [21] is an AMS framework, now in its third generation, intended to facilitate the development of analog integrated circuits. BAG could be used to craft a silicon compiler for FPGAs. It predates and is more featureful than BFG, but comes with stronger opinions about how layouts should be constructed and, crucially, is encumbered by restrictive licenses among its dependencies: BAGv2 requires access to the proprietary Innovus tool from Cadence, and BAGv3 requires access to the restrictively-licensed OpenAccess format. OpenAccess, an oxymoron, requires paid membership of the Si2 Consortium and is evidently not to be confused with "open access".

At least three new AMS frameworks are contemporaries of BFG. GDSFactory [100] was initially developed for photonics applications and is written in Python. GDSFactory now includes much of the functionality necessary to build an FPGA generator such as BFG, though questions remain about its scalability. Its benefits, on the other hand, are a relatively large (and professional) user base and the accessibility of Python, which we will discuss in Chapter 5.

Substrate [54] pursues similar goals to the BFG framework but is written in Rust. It initially targeted SRAMs and has already proven itself through tape-out of the SRAM22 [53] project. Unlike BFG, Substrate has a more unified view of the designer's workflow, for example by directly integrating interfaces to simulation tools. As will be seen in Chapter 2, BFG prefers a decoupled flow with many different point tools responsible for tasks that are not considered a core competency. Layout21 [30] is another Rust based IC layout framework, but one that integrates into VLSIR. Layout12 supports "raw" layout generation but also comes with more sophisticated ideas about how layout can be viewed programmed by a designer. GDSFactory, Substrate and Layout21 were all developed, or released, roughly in parallel with BFG. We consider the merits of switching between them in Chapter 5.

1.7.3 Open-Source FPGA Generators

In the early 2000s, Padalia et. al. [74] extended efforts in automating transistor-level layout of large systems to automatically generating FPGA circuit and layout based on various input architecture parameters. They used a set of fifteen hand-crafted cells (with custom layouts) as the inputs for placement, compaction and routing algorithms (together named GILES) that transformed a cell-level netlist describing a single FPGA tile into a final circuit and layout. The tile could then be repeated to form the fabric. The authors reported that, compared equivalent commercial devices, their tool produced layouts with 47% to 97% greater area. Kuon et. al. continued work in this vein [56], later augmenting it with an automatic transistor resizer [57] that optimizes the otherwise fixed cells used in this approach. Both Smith [95] and Chiasson et. al. [18] continued work on automatic transistor sizing with variations on algorithm and model, with the stated goal of expediting architecture exploration rather than automating circuit layout.

Not long before the work of Padalia et. al., commercial vendors began offering the first "embedded" FPGAs. These relied on digital VLSI flows (Sec. 1.3.2) to synthesise, place and route the FPGA fabric within a new or existing chip design. We call this *top-down* generation, because it focuses effort on the architectural space and the interaction with necessary tooling rather than on the physical design. Examples of early eFPGAs included the Actel VariCore, Leopard Logic HyperBlox FP, M2000 FLEXEOS, and eASIC 0.13 µm Core (see [46]). Academic efforts were introduced shortly thereafter and were steadily improved upon over the next 15 years. In 2003, Kafafi et. al. [46] reported a synthesised fabric with 540% overhead (over a comparable custom-designed commercial device). In 2017, Kim and Anderson [51] compared five eFPGA architectures with differing design goals to the then-contemporary Stratix III commercial FPGA. Compared to the Stratix, their designs occupied 1.3×, 1.5×, 1.9×, 2.5× and 2.9× more area. Though they admitted an imperfect comparison, they found that their designs had 1.8× to 11.7× worse delay too.

Before this, in 2008, Chaudhuri et. al. [16] compared an eFPGA fabric synthesised from standard cells to the output of the GILES tool from [56]. For a comparable architecture, they found that using the standard-cell methodology result in 16% greater area usage than using the GILES placer, compacter and router. They noted that a significant contributor to this difference was that they used foundry-provided flip-flop standard cells, whereas Kuon et. al has used smaller, custom SRAM cells for GILES. We can expect that with the same level of custom cells, the general-purpose tools would have achieved similar results. This seems to have heralded the shift over the next decade to mostly standard-cell-synthesisable fabrics: the GILES bespoke placement, compaction, and routing algorithms were already very similar to macro placement steps in regular VLSI flows, and as modern tools improved, the value of the FPGA-specific tooling seems to have diminished.

From 2019 to 2021, thanks in part to a surge of interest in (and funding for) open-source hardware projects, no fewer than three new open-source eFPGA generators were introduced: OpenFPGA [98], PRGA [62] and FABulous [52]. As our contemporaries, they share much of our view on the state of FPGA computing and also much of our motivation. Like other

eFPGA generators they start with a high-level description of the architecture and eventually emit an RTL (Verilog) netlist describing the entire fabric. What set this generation apart, however, was the end-to-end completeness of their ecosystems. Each project was able to *tape-out* (have manufactured) *usable* example FPGAs. Using increasingly complete open-source tools (Sec 1.7.1), they were able to generate architectural descriptions, benchmark their architectures, generate SPICE simulations, and program their fabrics.

To manage the VLSI flow, especially the placement and routing steps, modern eFPGA generators create sets of input constraints to help guide the VLSI tools. For example, they will typically emit RTL descriptions of components such as CLBs and switchboxes, expecting these to be synthesised, placed and routed as independent macros, and then provide a floorplanned (manual) placement of these macro blocks in a grid to recreate the regular FPGA grid structure. This has the practical and immediate benefit of simplifying the VLSI flow task and thus makes it faster to run and more likely to succeed. It also, to our view, serves as tacit acknowledgment of the straightforward regularity of fabric architectures and as a nod to how they should be built instead.

A fundamental problem with using digital design flows to create FPGA fabrics is that the circuitry required of them often breaks the digital abstraction. As a first example, consider that the interconnect wiring within an FPGA presents many combinational loops: paths whose logic is circularly dependent within a single clock cycle. In reality these circular paths are disabled by run-time configuration of the intervening switches, but static timing analysers don't know that and will simply balk at them instead. Embedded FPGA generators must take care [67] to avoid this problem, usually by both carefully arranging fabric connections, and instructing the tools to ignore particular paths.4

As a second example, consider that many established circuits for implementing FPGAs use analogue structures that are not readily supported by standard-cell libraries. Transmission gates are small but effective passive switches that are not easily accommodated by static timing analysis (because they don't include a buffer). Yet, they make effective programmable interconnection points in switchboxes and are also very area-efficient ways to build multiplexers. Pass gates are even smaller, comprising a single transistor, but require techniques such as gate boosting or substrate biasing to correctly pass low and high voltage signals. To work correctly, any boost or bias requires fine-grained tuning, and standard-cell libraries simply do not afford that level of control. Embedded FPGA generators eschew these structures altogether, or provide high level abstractions for their inclusion and defer the physical implementation to the user.

Put another way, the high-level abstractions used by such generators allow for the physical implementations of each component to be specialised as necessary. Though they start by targeting externally-produced libraries of standard cells, each flow includes a mechanism to replace the underlying circuitry and layout for components at various levels of the cell hierarchy. This is an extension of a standard feature of digital VLSI tools, which allow for external modules to be added into the flow. An alternative approach is to simply augment the standard library with new cells — that is, as long as the synthesis and technology mapping tools can be taught their purpose.

Piggybacking on digital design flows also has the advantage of being relatively *process portable*. That is, the same high-level architectural description and RTL code can be used to manufacture a design in more than one semiconductor process by "simply" re-running the appropriate tooling, standard cell libraries, rule sets, and so on, for each process.

Overall, eFPGA generators have two key advantages:

1. They enable the rapid implementation of architectural features. Because they rely on a high level of abstraction, most features need only be expressed as RTL logic with necessary metadata to fit in with other components. Since they readily tie into existing FPGA toolchains (Sec. 1.7.1), these features can be evaluated quickly through standard FPGA benchmark suites. They have already gained support for most of the most prominent architectural features described in FPGA literature.

2. They can be readied for manufacture by a small team. Since the user does not need to perform AMS physical design, a relatively common and automatic digital VLSI flow can be used to bring the design to manufacture. Indeed, these tools have already been validated through the manufacture of their designs by small teams of researchers and enthusiasts.

On the other hand, the main disadvantage of eFPGA generators has remained the performance of the resulting fabrics across achievable clock frequency, power consumption and/or logic density. The OpenFPGA authors reported [98] that, compared to a commercial device with an equivalent architecture and process node, their layouts incur a 60% area penalty and a 20% performance penalty. This represents a major improvement over the 540% area penalty reported by [46] about two decades prior, but is still prohibitive. Specialising key circuits ameliorates this deficit: the FABulous authors found [52] that merely substituting a custom 4:1 multiplexer cell into the standard cell library reduced the area of their fabric by 31%. Nonetheless, the overall performance gap remains.

1.7.4 Automatic Layout Generation

There is a significant body of work on the complete end-to-end generation of analogue circuits and layout. BonnCell [43], [104] uses traditional optimisation techniques for transistor placement and routing, notably with a mixture of Mixed-Integer Program (MIP), Integer Linear Program (ILP), Boolean Satisfiability (SAT) and custom methods. More recent advancements have been driven by Artificial Intelligence (AI)-based methods, which we differentiate from earlier numerical optimisation techniques by convention. Among a recent survey by the authors in [117], we find graph-neural-network based methods [24], [25]; Reinforcement Learning (RL) techniques [87]; Bayesian optimisation methods [63]; and hierarchical mixtures of multiple approaches [55]. Also recent, but contravening trend, VIPER [84] uses a classical controlled random-walk algorithm to generate DRC clean layouts in order to better model manufacturing defects.

The advent of generative LLM-based agents has heralded the possibility of fully-automated textual design specification. The authors in [58], [15], [31] and [114], have variously applied LLMs to topology generation, placement, routing and tuning. In [38] the authors invent a simple layout specification language to capture simple layout descriptions, and in [42] these techniques are applied to the generation of standard cells.

These works represent significant advancements towards fully-automated generation of analogue circuits and layout. Yet, despite the vast increase in available processing power that has enabled such computationally-intensive methods to succeed, the central disadvantage of these approaches remains scale. They are shown to work for dozens to a few hundred transistors, not thousands. Nonetheless, they are a promising and complementary approach to BFG's. They can be used to make the component cells within a larger silicon compiler, or applied to sub-problems within a larger design. We discuss this symbiosis in Section 5.1.9.

1.7.5 Open-Source CMOS Processes

1.7.5.1 SkyWater 130 nm Process

SkyWater 130 nm was the first *production* PDK to be released with an open-source license [93]. Prior to this, available open-source PDKs were largely *predictive* in nature. That is, they were mostly educated guesses about the properties that a process at a given technology node would *probably* have. Though invaluable to the research community and to students, these were not supported by any foundry, and could not be used for manufacturing a real device. FreePDK45 [96], FreePDK15 [9] and ASAP 7 [106] are prominent examples for the 45 nm, 15 nm and 7 nm nodes respectively. As can be seen, the advantage of predictive PDKs remains their availability for modern processes; SkyWater 130 nm is more than two decades old. Yet, SkyWater Technologies maintains SkyWater 130 nm production to this day, and the manufacturability of its designs makes the process a welcome initial choice for demonstrating our design methodology: first because it should enable us to compare real device performance to predictions, and second because it offers a tangible incentive for participation by the wider enthusiast community.

The SkyWater 130 nm PDK is documented by [93] in detail. We reproduce the semiconductor layer stack here for the reader's convenience and for reference throughout this document. Figure 1.4 shows a cartoon of the substrate, diffusion, polysilicon, metal and via layers that comprise the process, otherwise known as the *stack-up*. Table 1.1 summarises the internal name, a shorthand we will use for reference and a brief description of each of these layers.

1.7.5.2 Global Foundries 180 nm Process

Released a few years later, the Global Foundries 180 nm is a second PDK made openly available to all under a FOSS license. As we will see in Section 2.4, it bears enough similarity to the newer SkyWater 130 nm process that it their designs are cross-compatible, and subse-

Table 1.1: Commonly used SkyWater 130 nm layers and their shorthands

Layer	Shorthand	Description
`nwell.drawing`	w_n	N-type substrate well
`diff.drawing, psdm.drawing`	d_p	P-type diffusion region, inside N-type well
`diff.drawing, nsdm.drawing`	d_n	N-type diffusion region
`poly.drawing`	g	Polysilicon gate
`licon.drawing`	v_0	Via between polysilicon gate or diffusion and the local interconnect layer
`li.drawing`	m_0	Local interconnect layer ("metal layer 0")
`mcon.drawing`	v_0	Via between local interconnect and metal 1
`met1.drawing`	m_1	Metal layer 1
`via1.drawing`	v_1	Via between metals 1 and 2
`met2.drawing`	m_2	Metal layer 2
`via2.drawing`	v_2	Via between metals 2 and 3

Table 1.2: Commonly used Global Foundries 180 nm layers and their shorthands

Layer	Shorthand	Description
`nwell`	w_n	N-type substrate well
`nplus`	d_p	P-type diffusion region, inside N-type well
`pplus`	d_n	N-type diffusion region
`poly2`	g	Polysilicon gate
`contact`	v_0	Via between polysilicon gate, diffusion, and the local interconnect layer
`metal1`	m_0	Local interconnect layer
`via1`	v_0	Via between local interconnect and metal 1
`metal2`	m_1	Metal layer 1
`via2`	v_1	Via between metals 1 and 2
`metal3`	m_2	Metal layer 2
`via3`	v_2	Via between metals 2 and 3

quently serves as our first demonstration of process portability (Sec. 4.3.9). Again, detailed documentation can be found in [33]. We reproduce the Global Foundries 180 nm stack-up in Figure 1.5 and summarise key layers, with shorthand corresponding to the SkyWater 130 nm stack-up, in Table 1.2.

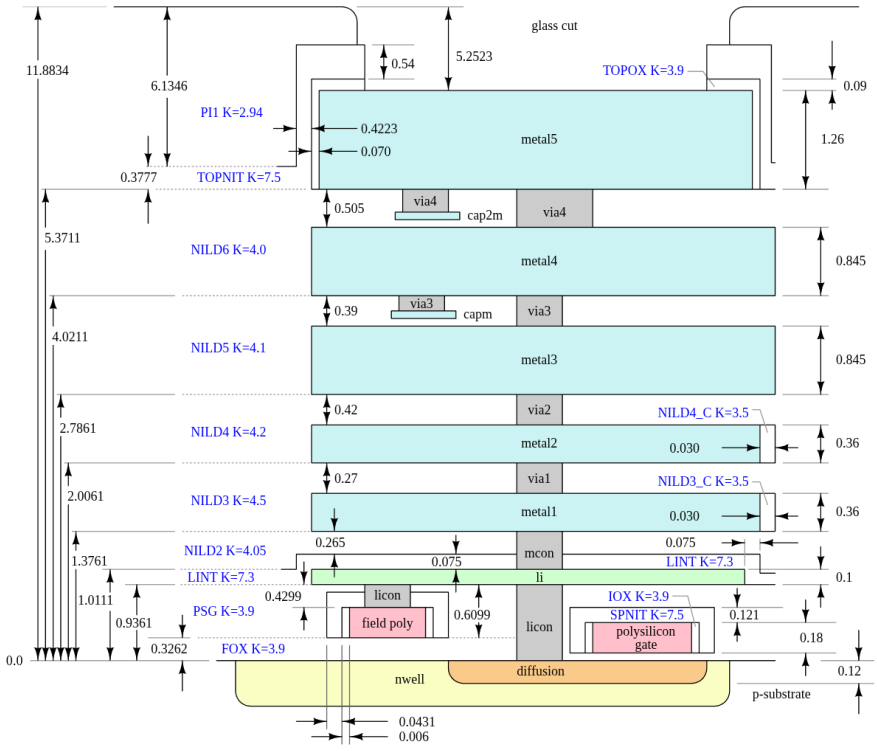


Figure 1.4: SkyWater 130 nm Process Stack Diagram
Adapted from [93].

1.8 Contributions

The contributions of this work are fourfold. All are free and open-source:

1. A C++ AMS framework for the programmatic generation of circuits and their layout. The framework encourages hierarchical composition and process portability, but is otherwise unopinionated, and supports a variety of utilities for the automation of design tasks. Using a middleware layer for the import and export of common industry formats, the framework can form the basis of an end-to-end AMS design flow or can be used to generate components for integration into a larger design. All of its dependencies are also free and open-source.

2. A parametrised generator for modern FPGA interconnect topologies based on the Xilinx UltraScale architecture. We reverse engineer the interconnect of a modern UltraScale device, identify its connectivity patterns, reason about likely circuit- and physical-level design decisions, parametrise the design so that it can be resized for our process node, and present algorithms for its construction.

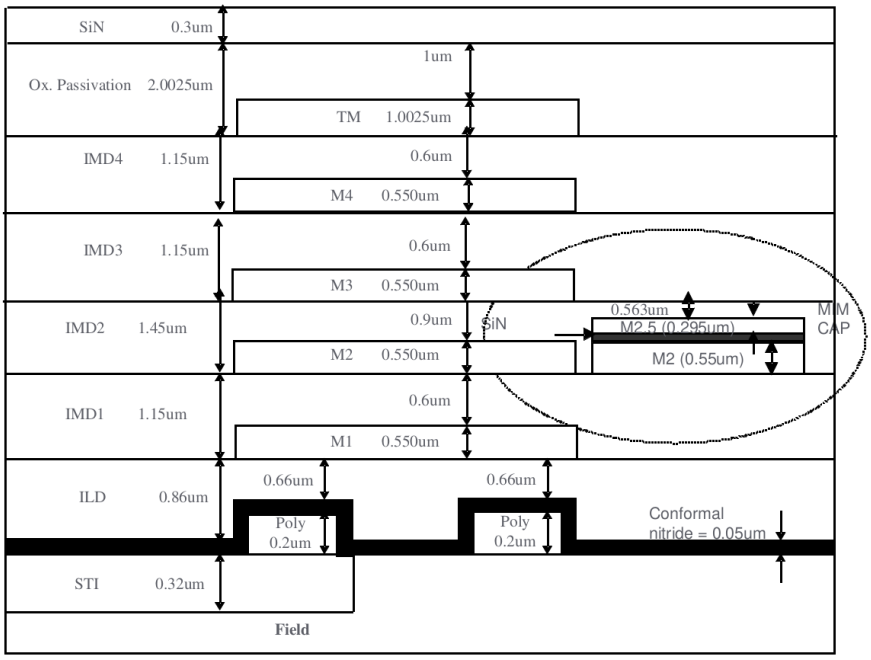


Figure 1.5: Global Foundries 180 nm Process Stack Diagram
Adapted from [33].

3. A repository of parametrised cell generators for FPGA components in SkyWater 130 nm. Basic generators build wire buses, transistors, simple logic and filler cells. Increasingly complex generators compose these types into FPGA fabric components such as CLBs, programmable multiplexers, and carry chains.

4. A modern columnar reconfigurable fabric architecture for SkyWater 130 nm that synthesises insights from (2) and key results from literature, together with a cell generator that demonstrates how (1), (2) and (3) can be combined to produce a complete FPGA fabric for this architecture.

1.9 Outline

The rest of this dissertation is organised as follows. In Chapter 2 we will describe how BFG enables designers to hierarchically compose circuitry and layout, how the tool supports process portability, and how it interacts with established IC design tooling. We discuss the required geometric primitives, data models, a series of convenient abstractions, BFG's automated router, and present usage examples. Then, in order to eventually show how BFG could be used to build a modern fabric in SkyWater 130 nm, we establish how modern fabrics have diverged from traditional architectures in Chapter 3. We describe how we reverse

engineered Xilinx's UltraScale interconnect and developed parametrised algorithms for its reconstruction. In Chapter 4 we catalogue BFG's collection of parametric generators for FPGA circuits in SkyWater 130 nm, and discuss the circuit, layout and software patterns behind each. In order to test our hypothesis, we then compare the performance of a BFG-generated 4-LUT CLB to one synthesised from standard cells. Next, we develop a modern columnar architecture based on the literature and our reverse-engineered topology generator, before demonstrating how BFG can be used to assemble a complete fabric for this architecture from a repeatable tile. Finally, we conclude and discuss future work in Chapter 5.

Chapter 2

BFG

2.1 An Analog/Mixed-Signal Generator Framework

To the circuit designer and FPGA architect, BFG is a C++ software framework for the generation of analog/mixed-signal CMOS integrated circuits. As discussed in Section 1.3.3, this means that BFG supports the broadest class of IC designs, along with the commensurate control and attention to detail that they require. At its core, the BFG library provides a data model and a set of software interfaces that enable designers to express the circuit topology and physical layout of their designs. Atop this library designers write *generators*, which are just unfettered C++ programs, to assemble a circuit (and its layout) according to the parameters they have elected to expose.

BFG is unopinionated about how a designer might choose to codify the generation of a circuit, but nonetheless provides a set of routines to aid the designer in the pursuit of various tasks. These range from geometric tests (such as collision detection), to straightforward layout aids (such as assembling components in rows), to more complex automation (the router). Sill, there are many software paradigms for expressing planar layouts, each best suited to one or more different objectives. In Chapter 4 we will sample some of these approaches through the collection of generators that BFG includes for FPGAs.

The validity of designs is the responsibility of the designer. For correct layout BFG provides convenient ways to access PDK-specific DRC rules, but at the time of writing it does not enforce or even report rule violations. Layouts must be exported to an external design tool for DRC checks. This was both a matter of expedience and trust: having the designer use trusted third-party tools to check their BFG work serves to validate BFG itself, otherwise BFG's DRC checker itself would need checking. Circuit correctness must likewise be verified by external tools for Layout-Versus-Schematic (LVS), that ensure the IC layout matches the given circuit description, and by SPICE simulations, that show whether the circuit behaves as intended.

In this chapter we will discuss the design of BFG, its circuit and layout models, its router and other conveniences, and how it interoperates with the rest of the EDA ecosystem from

Section 1.3.2.

2.2 Data Model

The fundamental unit of design in BFG is a *cell*. Cells, also known elsewhere as "modules" or "blocks", combine a circuit schematic together with an IC layout. A BFG *generator* is code that produces a cell. Generators take parameters that dictate features or other aspects of the cell, such as whether some sections are included, which inputs are exposed, transistor sizes, and so on. BFG expects designers to compose hierarchies of these parameterisable generators — that is, to build generators out of generators. This merely makes explicit how hardware is already built: sub-components are typically deferred to different teams or individuals with design requirements, assembled together. Those teams or individuals might in turn defer certain subcomponents to others. Likewise for BFG: because cell generators are C++ programs, they can use any manner of available tools, libraries, and other cells to produce their output cell.

As we will see in subsequent sections, hierarchical composition has two main benefits: 1) code reuse, amortising the effort to create each generator across every instantiation; and 2) abstraction. Point (1) is a fundamental cornerstone of software engineering. What then do we mean by abstraction? As one moves up the hierarchy of cells, one can be less and less concerned about the details that govern the layout of each, and more concerned about where child cells are placed with respect to one another and how they are connected together. This is what motivates BFG's router (Sec. 2.7): at a certain point up the hierarchy you only care about where cells are placed and not exactly how all their wires are routed together. You can also swap out which cells you use, depending on your parameters. Choosing the type of memory and multiplexer cells, for example, can be made to depend on the choice of memory interface and the number of inputs.

Hierarchical composition of cells also affords BFG its first mechanism for process portability. Instead of swapping out cells for sizing or memory type, designers can choose cells that are designed to support different manufacturing processes. That is, the process itself becomes a parameter. Among the FPGA cells we discuss in 4, the large interconnect multiplexers and CLB are good examples of cells which could usefully produce equivalent functionality for a variety of different processes. Unfortunately, this does not obviate the need to design technology-specific cells.

2.2.1 Technology Model

The foundries which develop (and sell access to) CMOS fabrication processes (Sec. 1.3.1) provide Process Development Kits (PDKs), which are packages of all data needed to produce valid designs, to their customers. These are the source of truth for any software needing to furnish the foundry with blueprints. BFG, and AMS tools like it, must model key aspects

of this data: the technology's layers ("stack-up") and how they interact, layout rules, and device models.

We start with the technology stack-up. It is typical for PDKs to label the technology's layers with a pair of integers: a "layer" and a "purpose". The layer portion usually, but not always, pertains to a physical step in building the technology stack like "metal layer one" (metal deposition) or "via two" (a cut-out); the purpose usually denotes that shapes are a "pin" (connection point), "label" (just names), or "regular manufacturable shapes". So that they can be used by humans, the foundry also provides names for these layers/purpose pairs, such as `met0.pin` to indicate a pin shape on the first metal layer. There are only rough conventions on what purposes exist, and even which layers are real and which are abstract. This is not BFG's problem to solve. BFG's internal model need only produce correctly-labeled information for the foundry at the end of the process.

Accordingly, BFG tries to capture the semantics of a PDK stack-up, make it readily accessible to designers and reproduce at design export. We map each process layer to a single number internally, simplifying storage and retrieval by this key. A separate mapping is maintained of human-readable labels for each layer. Users can query layer information by name (a string) or by number, though the latter is not usually available. The human-readable mapping is distinct because certain routines, such as those for automatic constraint checking, can elide the string look-up altogether. As with most layout-related information, these structures are gathered in the `PhysicalPropertiesDatabase` class.

Layer connectivity is stored in a separate structure which records the via layers that connect any two other layers. It is convenient, particularly when routing, to seek a path between shapes on two layers, so the `PhysicalPropertiesDatabase::FindViaStack` method can be used to find the shortest path through vias between the two layers. If we treat layers as nodes and vias connecting them as costed edges, find the shortest path is a straightforward application of Dijkstra's algorithm.

BFG also makes explicit the purpose of the *pin* layer, which is typically used to indicate to other tools that a given region within a layout can be used for electrically connecting shapes, such as by laying wires. LVS tools also use *pin* shapes to detect that a layout's ports match the circuit description. BFG tries to digest the meaning of pin layers by associating them with the layers they provide connectivity to, as shown below. This allows the designer to create ports on a layer without having to remember the requisite pin layer.

Listing 2.1: Pin semantics, excerpt from `physical_properties_database.h`

```
146 class PhysicalPropertiesDatabase {

295   // For a given pin layer, find the layers which can access it. The pin layer
296   // represents access to a given layer, which is the first entry. For each of
297   // those we have a set of layers which can be accessed through one more vias.
298   const std::vector<std::pair<geometry::Layer, std::set<geometry::Layer>>>
299   FindReachableLayersByPinLayer(const geometry::Layer &pin_layer) const;

301   const std::set<geometry::Layer> GetPinLayersFor(
302       const geometry::Layer &layer) const;

304   // For a given layer, returns:
```

```
305   // — the layer
306   // — any layers for which the layer is an access pin layer
307   // — any layers which can access any of these layers with a via.
308   const std::set<geometry::Layer> GetAccessibleLayersForPin(
309         const geometry::Layer &layer) const;

311   bool IsPinLayer(const geometry::Layer &layer) const;

386 };
```

BFG encourages the use of layer *aliases*, which provide alternate human-readable names for existing layers. To see why, consider that it is common for foundries to use a single layer where others might distinguish several. For example, as alluded to in Table 1.1, SkyWater 130 nm's one `licon.drawing` layer is used to connect to polysilicon *or* diffusion from the local interconnect layers. There is never ambiguity because the layout rules explicitly forbid situations where there might be ambiguity. By defining an alias for each specific case, users take one small step to unshackling themselves from the tyranny of the specific PDK they are working in. More broadly, this helps portability of their code to other PDKs in which the distinctions *are* made. In the same vein, aliases can also be used to create a common set of layer names that apply to multiple PDKs, as in the following excerpt mapping equivalent layers from the SkyWater 130 nm namespace to the Global Foundries 180 nm one:

Listing 2.2: Layer aliasing, excerpt from `dev_pdk_setup.cc`

```
320 void SetUpGf180Mcu(bfg::PhysicalPropertiesDatabase *db) {
374   db->AddLayerAlias("diff.drawing", "comp.comp");
375   db->AddLayerAlias("nsdm.drawing", "nplus.nplus");
376   db->AddLayerAlias("psdm.drawing", "pplus.pplus");
377   db->AddLayerAlias("poly.drawing", "poly2.poly2");
378   db->AddLayerAlias("nwell.drawing", "nwell.nwell");
379   db->AddLayerAlias("li.drawing", "metal1.metal1");
380   db->AddLayerAlias("met1.drawing", "metal2.metal2");
381   db->AddLayerAlias("met2.drawing", "metal3.metal3");
382   db->AddLayerAlias("met3.drawing", "metal4.metal4");
383   db->AddLayerAlias("licon.drawing", "contact.contact");
384   db->AddLayerAlias("mcon.drawing", "via1.via1");
385   db->AddLayerAlias("via1.drawing", "via2.via2");
386   db->AddLayerAlias("via2.drawing", "via3.via3");
```

Similarly, there are cases where PDKs use combinations of layers to signify different physical structures. In SkyWater 130 nm the combination of layers `diff.drawing` ("diffusion") and `psdm.drawing` ("P-type doping") creates a P-type diffusion region, as would be needed for a PMOS transistor. A corollary of this is that a single layer can be used for many different purposes. Here `diff.drawing` means both P- and N-type diffusion depending on the context. Aliasing can be used to indicate that a single layer has many uses, but BFG does not currently provide shortcuts to combinations of layers. This is because, usually, a shape on one of the layers in the necessary union is drawn around several others shapes on the other layer by a generator further up the hierarchy. An abstraction for this is technically feasible and worth doing in future.

Layout Rules

Each PDK provides a large set of layout rules that designers must follow. This is so that the foundry may reasonably guarantee that their designs will function as intended, given the tolerances of their equipment. As the physical features of these technologies have gotten smaller, the necessary manufacturing techniques have gotten more complex, and so too have the sets of rules. It is intractable to comprehend the rules of every process with the goal of building a general model, even though such a thing would be very useful. Instead, BFG captures the *classes* of rules needed to build correct layout as they are encountered in each process.

We distinguish broadly three categories of layout design rules:

1. intra-layer, governing sizing and spacing of shapes on a single layer;
2. inter-layer, governing sizing and spacing of shapes between layers; and
3. global rules.

Global rules are usually not handled explicitly by designers in their generators. The most prominent example is the grid-alignment requirement: all shape edges and corners must fall onto a regular grid with a pitch defined by the manufacturer. For SkyWater 130 nm, this grid has a pitch of 5 nm and no offset, so all physical coordinates must be in multiples of 5 nm. BFG accommodates this by using separate internal and external coordinate systems. Real, physical coordinates are mapped to an integer-valued internal coordinate system where any fractional values are truncated to the lower-bounding integer. As a convenient consequence of integer arithmetic in C++ and languages like it, this should present minimal computational overhead. Internal values are mapped back into the external (physical) coordinate system when designs are exported.

For example, in Skywater 130 nm the designer sets the ratio of internal units u (not to be confused with micrometres) to external units at $1u = 5\,\text{nm}$. The designer can then marshal values back and forth with the aid of two helper functions, wherever necessary:

```
void ToyWithUnits(const PhysicalPropertiesDatabase &db) {
    db.set_internal_units_per_external(0.2);
    int64_t x_int = db.ToInternalUnits(7);      // 7 nm -> 1 unit
    int64_t x_ext = db.ToExternalUnits(x_int);  // 1 unit -> 5 nm
}
```

For intra- and inter-layer rules, BFG defines the rule structures `IntraLayerConstraints` and `InterLayerConstraints`, respectively, which are reproduced (and simplified) in Listing 2.3. As a matter of convention, variables storing values in external units are suffixed with their physical units, such as `pfet0_width_nm`, which is in nanometres. The fields in these structs are in internal units.

Listing 2.3: Constraint structs, sanitised excerpts from `physical_properties_database.h`

```
struct IntraLayerConstraints {
  int64_t min_separation;   // All of these are in internal units!
```

```
  int64_t min_width;
  int64_t min_pitch;
  int64_t min_area;
  int64_t min_hole_area;
  int64_t max_width;
  int64_t via_width;
  int64_t via_height;
  double via_cost;
};

struct InterLayerConstraints {
  int64_t min_separation;
  int64_t max_separation;
  int64_t min_enclosure;
  int64_t min_extension;
  int64_t min_enclosure_alt;
  int64_t via_overhang;
  int64_t via_overhang_wide;
};
```

Within a layer, we find it sufficient to record:

A. the minimum separation between shapes;

B. the minimum and maximum widths (size in either horizontal or vertical dimension);

C. the minimum area of any shape;

D. the minimum area of holes in shapes; and

E. the size of any vias (for via layers).

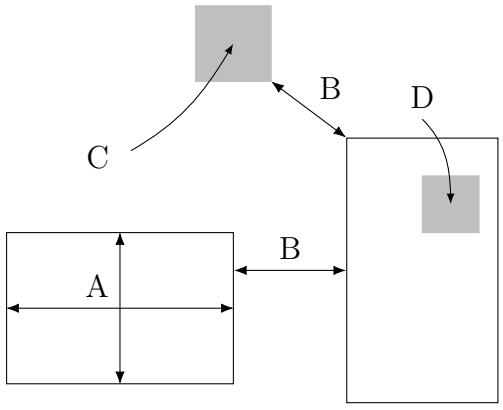


Figure 2.1: Some intra-layer rules

Refer to Figure 2.1 for visual examples of some of these. The minimum pitch `min_pitch` is a shortcut to the sum of width and separation to use when placing wire tracks for routing

(Sec. 2.7). The `via_cost` figure is also used by the router to determine the penalty for a path using a via on that layer.

Referring now to Figure 2.2, *between* layers we include rules that dictate:

A. the minimum and maximum separation of shapes on the two layers;

B. the minimum and maximum enclosure of shapes on one layer by another; and

C. the minimum enclosure of shapes on via layers by the metal layers they connect, which tend to distinguish a short and long dimension.

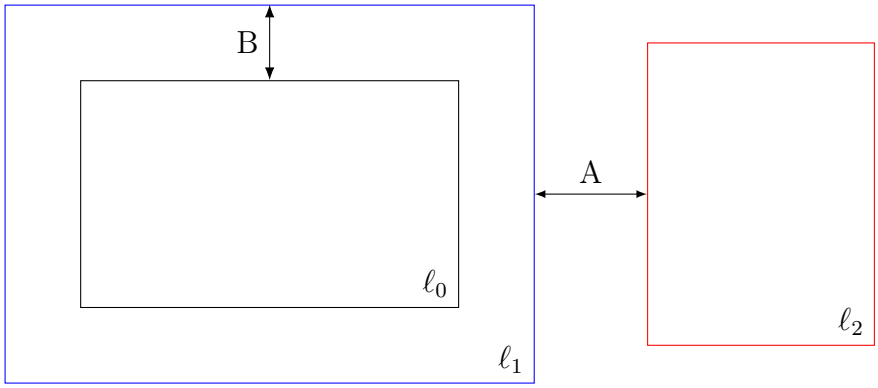


Figure 2.2: Some inter-layer rules

Constraints are collected together in the `PhysicalPropertiesDatabase` class, which exposes various `Rules(...)` methods to perform look-ups. Intra- and inter-layer rules are stored in maps with a single and a dual tree structure over the keys, respectively. Consequentially, rule look-ups are not ordered. This is to aid the designer by not forcing them to remember which way around a particular relation was stored. For example:

```
// Intra-layer lookup.
const auto &met1_rules = db.Rules("met1.drawing");
int64_t gap = met1_rules.min_separation;

// Inter-layer lookups.
const auto &met1_via_rules = db.Rules("met1.drawing", "via1.drawing");
const auto &via1_met1_rules = db.Rules("via1.drawing", "met1.drawing");

// Both retrieve the same struct.
assert(met1_via_rules == via1_met1_rules);
```

However, rule relations between layers are not commutative: minimum protrusion by layer ℓ_0 over layer ℓ_1 is not the same as protrusion by layer ℓ_1 over ℓ_0. There are cases where both are defined. For example, Figure 2.3 shows a typical scenario where polysilicon gates g must protrude past diffusion layers d_n and d_p (A), but separately each diffusion layer must protrude past g in the MOSFET source and drain (B). Because BFG stores both together,

there must be a separate field in the struct stored at (g, d_n) that can differentiate protrusion (g, d_n) versus (d_n, g). In this case we call one the `min_extension` and the other `min_enclosure`. In other cases we admit `min_enclosure_alt` and use it and `min_enclosure` together. This is a workable but confusing solution; where possible we rely on informative documentation to guide the designer as for example in Listing 2.4. Perhaps it is better to store rules relationally, since the designer is burdened to remember the directionality either way. Or perhaps the lookup system could be more descriptive and, therefore, more intuitive. We leave that to future work (Ch. 5).

Listing 2.4: Documenting rules against confusion, from `dev_pdf_setup.cc`

```
bfg::InterLayerConstraints inter_constraints = {
  //  poly.4: (min) "Spacing of poly on field to diff (parallel edges only)"
  .min_separation = db->ToInternalUnits(75),
  //  poly.8: (min) "Extension of poly beyond diffusion (endcap)",
  .min_enclosure = db->ToInternalUnits(130),
  //  poly.7: (min) "Extension of diff beyond poly (min drain)"
  .min_extension = db->ToInternalUnits(250)
};
db->AddRules("poly.drawing", "pdiff.drawing", inter_constraints);
db->AddRules("poly.drawing", "ndiff.drawing", inter_constraints);
```

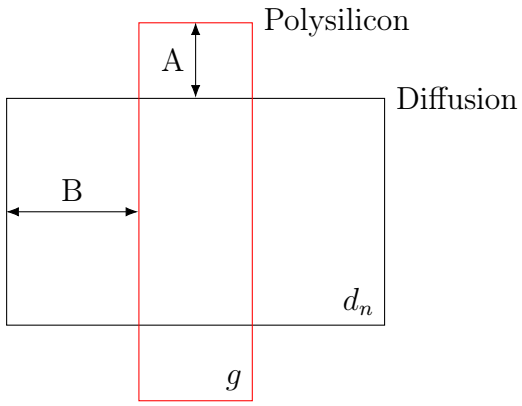


Figure 2.3: Non-commutative DRC rules

More generally, we observe that CMOS technologies can be grouped into classes of similar layout topologies (styles). Layouts should be broadly reusable for processes within a particular class by making changes only to spacing and sizing. The SkyWater 130 nm (Sec. 1.7.5.1) and Global Foundries 180 nm (Sec. 1.7.5.2) processes, around which BFG was initially built, exist in one such class. Both are decades-only process, *planar* and *single-patterned*: unlike in modern processes, poly-silicon structures can be spaced unevenly and even broken, and transistor sizes are not quantized by some number of fins.

This is, incidentally, BFG's second mechanism for process portability. As long as 1) generators are written with abstract rule look-ups and no hard-coded values and 2) there is a straightforward mapping of layers between two technologies, it is trivial to produce a

layout for a second technology using a generator by simply swapping rules and providing a layer mapping. This will work as long as the designed and the intended technologies share a layout class. An example will be shown in Chapter 4: the `Gf180McuMux` mutliplexer is a 4:1 multiplexer circuit for the Global Foundries 180 nm process (Sec. 4.3.9) that is made by re-applying the `Sky130Mux` generator originally designed for the SkyWater 130 nm process (Sec. 4.3.8).

We have already seen how certain useful rules can derive from the base set of recorded constraints with the "synthesised" `min_pitch` field for intra-layer rules. This was itself an experiment. In fact, there are many rules that derive from the base set and that are frequently used in layout code. A general mechanism for these might be necessary, but we prefer not to pollute the base structs with such ad-hoc additions. Some languages provide built-in mechanisms to do this, such as *extensions* in the Swift Programming Language. So far BFG has opted to support each only as the need arises. One such need is in determining the metal encapsulation of vias.

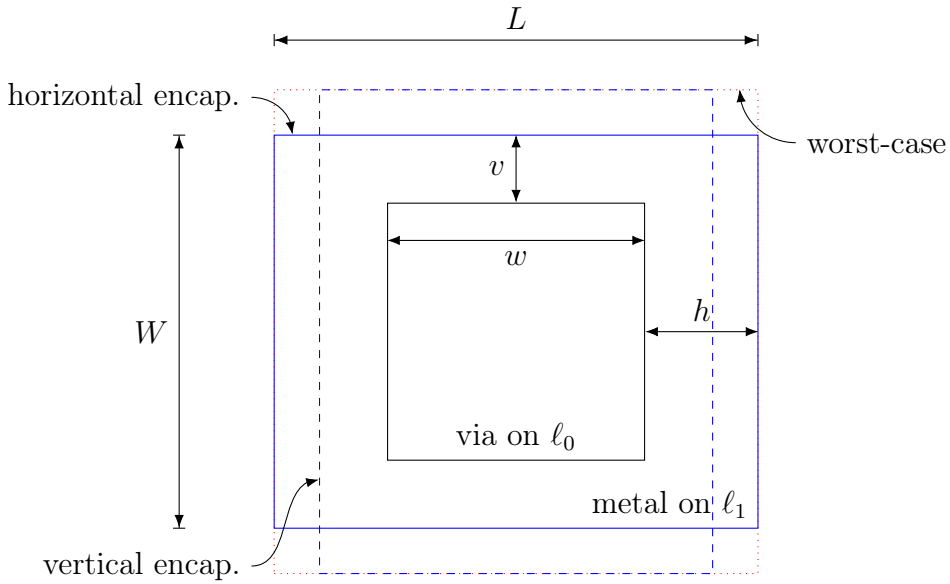


Figure 2.4: Synthesised rules: via encapsulation

The minimum-sized encapsulation of a square via by metal is defined by the width of the minimum width of the via w and the minimum enclosure of the via horizontally h and vertically v (per Figure 2.4). SkyWater 130 nm rules do not distinguish the two directions, so the resulting rectangle depends on the direction of the wire. To save designers having to write this every time, BFG includes helper methods `PhysicalPropertiesDatabase::TypicalViaEncap`. The first variation considers only the encap from the below or above the via. The second variation finds the bounding box of the union of both, which is the worst-case, since both adjoining layers need be considered when spacing vias apart. BFG synthesises the overall

width W and length L of the encapsulation metal over the via (with respect to the direction of the wire nominally arriving at the via) for convenience:

Listing 2.5: Helpers for sizing typical via encapsulation by metal, excerpt from `physical_properties_database.h`

```
106 struct ViaEncapInfo {
107   // Size of the encap in the direction of the wire.
108   int64_t length;
109   // Size of the encap in the direction across the wire.
110   int64_t width;

112   int64_t MaxSide() const {
113     return std::max(length, width);
114   };
115   int64_t MinSide() const {
116     return std::min(length, width);
117   }
118 };

146 class PhysicalPropertiesDatabase {

259   // This convenience function returns a rectangle in the shape of a
260   // typically- (or minimally-)sized via-encap for the given layer.
261   ViaEncapInfo TypicalViaEncap(
262       const std::string &encap_layer_name,
263       const std::string &via_layer_name) const;
264   ViaEncapInfo TypicalViaEncap(
265       const geometry::Layer &encap_layer,
266       const geometry::Layer &via_layer) const;

268   // This convenience function returns the size of an encap that has to cover a
269   // via above and below:
270   ViaEncapInfo TypicalViaEncap(
271       const std::string &bottom_via_layer,
272       const std::string &encap_layer,
273       const std::string &top_via_layer) const;

386 };
```

Circuit Primitives

In addition to physical properties and layout rules, each PDK must also provide device models that are tuned by the manufacturer to represent the behaviour of transistors in that particular technology. Accordingly, BFG takes as input a list of primitive device models available in the PDK so that the designer can instantiate these in circuit descriptions. This is the purpose of the `CircuitPrimitivesDatabase` class:

Listing 2.6: Circuit primitives database interface, from `circuit_primitives_database.h`

```
19 class CircuitPrimitivesDatabase {
20  public:
21   CircuitPrimitivesDatabase() = default;
22   void LoadTechnology(const vlsir::tech::Technology &pdk);
23   void LoadPackage(const vlsir::circuit::Package &package);
24   Circuit *GetCircuit(const std::string &name) const;
```

```
25   std::string Describe() const;

27  private:
28   std::vector<std::unique_ptr<Circuit>> circuits_;
29   std::unordered_map<std::string, Circuit*> circuits_by_name_;
30 };
```

As with physical and layout rules, circuit primitives are loaded at start according to arguments provided by the user. They are typically retrieved (from disk or elsewhere) in the form of a VLSIR `vlsir.circuit.Package` Protocol Buffer message, which will be discussed in Section 2.8.1.

2.2.2 Circuit Model

The BFG circuit model is designed to support the circuit models of both SPICE simulators and higher-level RTL languages. Circuit behaviour is canonically described by the former, which typically assign single-width signals to each node in the circuit. RTL languages, on the other hand, allow one to specify signal *buses*, which have width greater than one, and that can be aliases for groups of signals from one or more other buses. The additional expressivity is useful for human designers reasoning about complex digital logic: re-grouping and renaming signals attaches semantic meaning that would otherwise be lost. BFG's data model supports both circuit views so that, should it be necessary, this additional information is not lost and generated circuits can be mapped to higher-level RTL logic descriptions[1].

The fundamental unit of circuit design in BFG is the `Circuit` class, which describes every component in a circuit and how they are connected, the circuit's external ports, and circuit-wide parameters. Every instance of a `Circuit` in another is represented by an `Instance` object, wherein the instantiated circuit is referred to as the *parent module* or *template circuit*. `Instance`s record which signals are connected to each of the parent module's ports, as well as any parameters defined for that particular instantiation, which in turn override circuit-wide parameters defined in the parent. An external circuit (whose internal components and connections are unknown, but whose port names and order are necessary) is also represented by a `Circuit` object, and can be indicated by setting the internal `type_` flag to `bfg::Circuit::-Type::EXTERNAL`, which is mostly useful to track in order to indicate as much to downstream tools.

A circuit's input and output ports are named terminals with associated internal signals. They are represented by `Port` objects, which additionally store directionality (input, output, both, or none). In AMS designs most ports are input *and* output; the distinction is most useful for digital logic. To record a connection to one of its `Port`s, an `Instance` stores `Connection` objects that discriminate between the types of connectable entities: `Slice`s, `Signal`s, and `Concatenation`s.

[1]Note that, at the time of writing, BFG's RTL-level model is incomplete, mostly because support is lacking for aliasing signals and concatenations. The missing elements are documented through-out the codebase. See the discussion of BigSpicy in Section 2.8.2 for a completed open-source implementation instead.

BFG uses *qualified* naming for circuits (and layouts), meaning that elements are named with a pair of strings denoting their domain name and common (domain-local) name. Domains provide separate namespaces for common names. This is a common practical consideration. For example, Java namespaces, which separate cell namespaces, ensure that independent developers do not inadvertently cause naming conflicts outside of their own domain. Qualified names are persisted through import and export with VLSIR (Sec. 2.8.1).

The `Signal` class represents circuit nodes. Signals are defined with a *width*. Buses are modeled as signals with widths greater than one. Signal names must be unique within the circuit they are defined in and, accordingly, are always owned by a single `Circuit` object. The `Slice` class exists as a light-weight reference to nodes within a `Signal`. In AMS design it is common to refer to single-width signals, so we also create `Wire` as a special case of `Slice` that only references a single node:

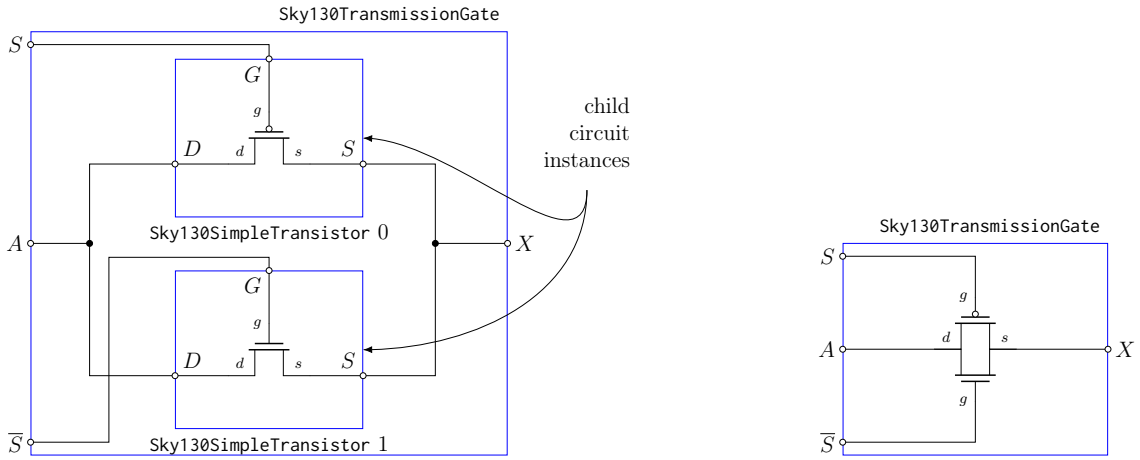
Listing 2.7: `Slice`s are light-weight references to `Signal`s, from `circuit/slice.h` and `circuit/wire.h`

```
class Slice {
 public:
  static Slice FromVLSIRSlice(
      const Circuit &circuit,
      const vlsir::circuit::Slice &slice_pb);
  Slice(const Signal &signal, uint64_t low_index, uint64_t high_index);
  Slice(const Slice &other) {}

  ::vlsir::circuit::Slice ToVLSIRSlice() const;

  uint64_t Width() const { return high_index_ - low_index_ + 1; }

  const Signal &signal() const { return signal_; }
  uint64_t low_index() const { return low_index_; }
  uint64_t high_index() const { return high_index_; }

 private:
  const Signal &signal_;
  uint64_t low_index_;
  uint64_t high_index_;
};

class Wire : public Slice {
 public:
  Wire(const Signal &signal, uint64_t index)
      : Slice(signal, index, index) {}
};
```

In order to support an RTL-level circuit model, a `Concatenation` class also exists as a combined reference to the concatenation of several other `Signal`s, `Slice`s and `Concatenation`s, though it has not been exercised and is considered incomplete. Consider the following Verilog statements:

```
wire a, b;
wire [2:0] c;
assign d = {c[2], b, a, c[1:0]};
```

BFG models the Verilog intuitively. A `Concatenation` "d" would join a `Wire` into signal `c` at index 2, single-width signals `b` and `a`, and a `Slice` into `c` from index 0 to 1.



(a) Before flattening

(b) After flattening

Figure 2.5: Overview of the flattening operation provided by `Circuit` class

The `Circuit` class provides a `Circuit::Flatten` operation as a convenience to the designer. *Flattening* a circuit removes one layer of the circuit hierarchy by merging in the implementations of all contained instances (though it is more commonly known as *elaboration* in typical VLSI flows). This gives the designer the freedom to compose a circuit with additional layers of hierarchy (by creating sub-generators) and to then simplify that hierarchy to either match the layout or to simply yield a more tractable output. As an example, consider a generator for a compound structure that relies on individual transistor generators: the transistor generators are likely to wrap a primitive transistor device in the PDK, but this level of indirection is ultimately superfluous. Flattening removes this indirection and restores the underlying primitives to the top-level circuit description. We use the example of `Sky130TransmissionGateStack`, which uses `Sky130SimpleTransistor` instances, to illustrate this in Figure 2.5. (Both of these generators are discussed in Chapter 4.)

In order to flatten a circuit, we copy the implementation from the template module of each child instance into the top-level circuit. Care must be taken to keep the names of signals being copied unique in order to prevent inadvertent connections. To achieve this, the name of the merged-in instance is used as a string prefix for the signal name. At circuit boundaries, internal port signals are simply replaced ("re-written") with the connecting signals from the parent circuit. Ports are then discarded, since the boundary they straddle has vanished. In order to prevent the prefixing operation from mangling names that are deliberately common to both parent and child circuits, one can define *global* signals that are exempt from prefixing. These typically include the power, ground and clock signals. Since

each call to `Circuit::Flatten` removes one layer of hierarchy, it must be called repeatedly until the cell has no child instances remaining in order to completely flatten a circuit.

2.2.3 Layout Model

Like its circuits, BFG composes layouts hierarchically. The fundamental unit of layout design is an object of the `Layout` class. It contains:

- collections of shapes, organised by layer;
- `Port`s, which are the physical analogue of `circuit::Port`s and are used to connect to external wiring; and
- `Instance`s, which like their circuit analogues, represent an instance of a design defined in another `Layout` object elsewhere.

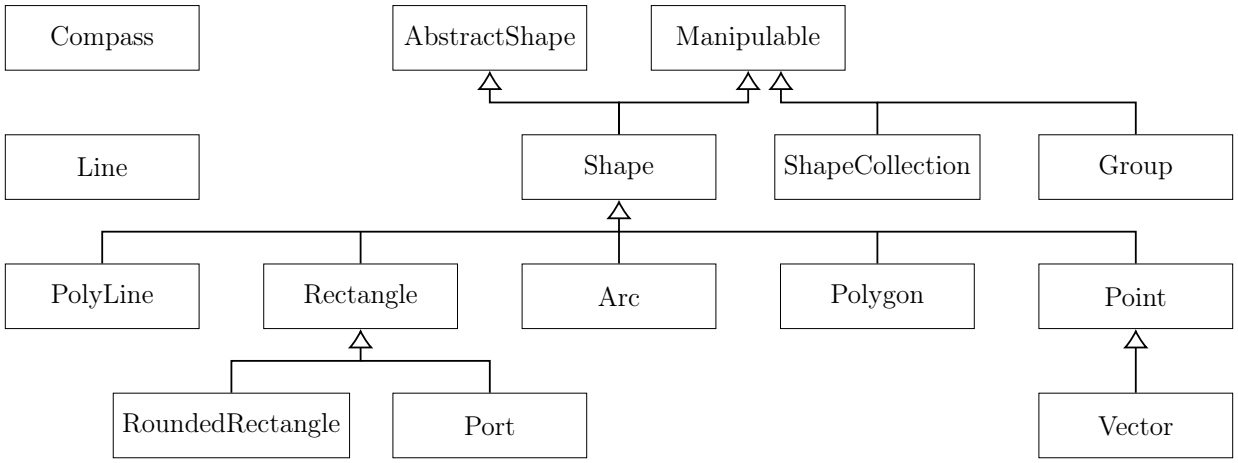


Figure 2.6: Geometric types

A simplified overview of BFG's geometric types is shown in Figure 2.6. They are:

- `Point`, a pair of integers representing $(x,\ y)$ Cartesian coordinates. As discussed in Section 2.2.1, BFG purposefully uses an integer-valued coordinate system with an external conversion factor to ensure trivial alignment to foundry-specified grids uses internal units.
- `Vector`, interchangeable with `Point`, an alias created to explicitly indicate when the coordinate is being treated as a vector from the origin.
- `Polygon`, a simple polygon. BFG assumes that polygons have no holes and no self-intersections.

- `Rectangle`, a rectilinear rectangle, defined by lower-left and upper-right `Point`s.
- `RoundedRectangle`, a rectangle with rounded corners. These are useful for hit-testing around draw `Rectangle`s, and not for use in layout themselves.
- `Line`, representing a straight line through two `Point`s, often used for geometric tests.
- `PolyLine`, a sequence of `Line`s that can be automatically inflated to a certain width. These are similar to *paths* in other AMS tools.
- `Arc`, the arc of a circle centred at some `Point`, also often used for geometric tests.
- `Compass`, an enumeration of the cardinal directions north, east, south and west, as well as the in-between positions. `Compass` also codifies the fact that in all BFG layouts these directions are equivalent to up, right, down, and left, respectively.

All shapes are subclasses of the `Shape` type which, in a rare display of Object-Oriented Programming (OOP), derives from the abstract classes `AbstractShape` and `Manipulable`. `AbstractShape` associates with the object a string "net" label, a boolean flag to indicate if the shape can be connected to by other shapes, and a layer number. It is named for the concept of a physically-realisable shape in the layout rather than the OOP pattern (though it is that too). The net label identifies a common node in the circuit, so that all shapes with the same net label are considered electrically connected. The router uses net labels and the `is_connectable` flag to determine which routes can connect to which shapes.

`Manipulable` provides a consistent interface for geometric transformations but defines none. Each shape maintains its boundaries and position differently, and so each shape must provide implementations of the common transformations shown in Listing 2.8.

Listing 2.8: Common transformations for each shape defined by `Manipulable`, from `geometry/manipulable.h`

```
14 class Manipulable {
15  public:
16   // Mirror in the y-axis (x = 0).
17   virtual void MirrorY() = 0;
18
19   // Mirror in the x-axis (y = 0).
20   virtual void MirrorX() = 0 ;
21
22   // Mirror in place about the central vertical axis.
23   virtual void FlipHorizontal() = 0;
24
25   // Mirror in place about the central horizontal axis.
26   virtual void FlipVertical() = 0;
27
28   virtual void Translate(const Point &offset) = 0;
29
30   virtual void MoveLowerLeftTo(const Point &point) {
31     ResetOrigin();
32     Translate(point);
33   }
```

```
34
35    // Rotate the object counter-clockwise about the origin; units in degrees.
36    virtual void Rotate(int32_t degrees_ccw) {};
37
38    // Translate the shape such that the lower-left point of the bounding
39    // box sits at the origin (0, 0).
40    virtual void ResetOrigin() {}
41  };
```

Of the geometric primitives, only the `Rectangle` and `Polygon` are needed for drawn layout, though in principle this is only limited by the export format. (Notably, there is currently no support for exporting arcs or paths.) `Port`s are just `Rectangle`s with clearer semantics around the associated net label during construction. `PolyLines` are inflated into `Polygon`s, but it is often convenient to keep them in their original form so that, for example, via bulges can be added by a later function. Together, these four types are the only shapes BFG needs to store in order to build a layout. Most of the other shapes are ephemeral: used for computation but rarely kept.

`ShapeCollection` and `Group` both provide a mechanism for grouping `Rectangle`, `Polygon`, `Port` and `PolyLine` objects. They both support group transformations across the whole collection by implementing the `Manipulable` interface. The difference is in the ownership of the underlying objects: `ShapeCollection` owns the shapes it contains, but `Group` does not (it keeps pointers to the shapes). As an example, a `Layout` stores all of its shapes in `ShapeCollections` because it must own them, but methods that produce or select some subset of these shapes will return references to them in a `Group`.

Just like their circuit analogues, `geometry::Instance`s represent an instance of a template layout defined elsewhere. Since they are just references, `Instance`s cannot manipulate the layout directly and must record any applied transformations. Each instance stores the position of referenced layout's origin, any applied rotation (counter-clockwise in degrees), and whether the instance has been mirrored in the horizontal ($y = 0$) axis. The combined `Manipulable` transformations are reduced to just rotation and y-axis mirroring to simplify storage, matching the schema used by VLSIR. (Translation simply moves the origin.) The exception to this mechanism is in generating instance-specific `Port`s, which must store their final post-transformation positions in order to simplify use by client code (such as the router, see Section 2.7). Ports are generated lazily by copying the ports from the template layout and computing any applicable transformations. If the instance is transformed again, the generated ports are invalidated.

While it is common for an instance of another circuit in the circuit definition to be matched by an instance of its layout from the same `Cell`, this is not a requirement. The circuit and layout hierarchies can be completely independent; you can instantiate a cell's circuit when assembling a circuit model but instantiate that of a different cell in the layout model. This affords the designer some flexibility so long as they ensure that the layout and circuit descriptions ultimately correspond.

Next, consider that it is natural to organise the generation of shapes into structures with shared purpose. These structures will typically require several shapes across several

layers — creating a wire, for example, usually requires drawing shapes on metal layers and intervening via layers. It is most intuitive to arrange the code by vertices of the wire path, not according to shape layers. Put another way, organising by layer is rarely useful. To facilitate this, `Layout` objects maintain a stack of most-recently active layers; any shapes added to the layout (through `Layout::AddRectangle`, `Layout::AddPolygon`, and so on) are automatically assigned to the currently active layer. See Listing 2.9. The caller can push new layers onto the stack through `Layout::SetActiveLayerByName`, add shapes, and then revert back to the previous layer (without knowing what it was) by calling `Layout::RestoreLastActiveLayer`. BFG also provides the `ScopedLayer` class to simplify this interface through the Resource Aquisition Is Initialisation (RAII) programming idiom (Listing 2.10).

Listing 2.9: Active layer stack

```
layout->SetActiveLayerByName(
    "via1.drawing");
layout->AddRectangle(
    geometry::Rectangle::CentredAt(
        {0, 0}, 10, 10));
layout->RestoreLastActiveLayer();
```

Listing 2.10: RAII with `ScopedLayer`

```
{ // Local block scope.
  ScopedLayer sl(layout, "via1.drawing");
  layout->AddRectangle(
      geometry::Rectangle::CentredAt(
        {0, 0}, 10, 10));
} // Destructor ScopedLayer::~ScopedLayer
  // restores last layer.
```

When crafting geometry, it is intuitive to create reference points. These points can be used to expose intermediate computations, provide anchors for related layout, provide connection points, and so on. When laying out a transistor, for example, it is convenient to create reference points to annotate where a via could legally be placed for connection to the diffusion or polysilicon shapes, without having to create a formal `Port`. The `Layout` class allows the designer to name and persist reference points through `Layout::SavePoint`. The points can then be recalled with `Layout::GetPoint` and `Layout::GetPointOrDie`. Listings 2.11 and 2.12 show an example of how a reference point set by a child layout can be used by its parent to guide the alignment of a routing grid.

Listing 2.11: Saving a grid alignment anchor in `tiles/lut_b.cc`

```
// Set the grid alignment point to fall on
// the output port of this memory:
std::vector<geometry::Port*> q_ports;
memories_[0]->GetInstancePorts(
    "Q", &q_ports);
layout->SavePoint(
    "grid_alignment_point",
    q_ports.front()->centre());
```

Listing 2.12: Using a saved anchor to guide grid alignment

```
auto alignment_point = lut_b->GetPoint(
    "grid_alignment_point");
if (alignment_point) {
  LOG(INFO) << "Aligning grid to "
            << *alignment_point;
  RoutingGridGeometry::AlignRoutingLayerInfos(
      *alignment_point,
      &met1_layer_info,
      &met2_layer_info);
}
```

Several other routines are provided as conveniences for the designer, chosen for their frequent use throughout the development of the generators in Chapter 4. As the software matures we expect that the most useful of these conveniences, and a host of new ones, will form the `Layout` interface. The suitability and *usability* of such interfaces will be discussed

the Chapter 5. The additional conveniences are:

1. a `Layout::Flatten` operation that operates analogously to `Circuit::Flatten`, copying one layer of instantiated layouts from their templates into the current cell while prefixing shape's net labels (unless they are global nets) and any saved reference points;
2. two operations to get the layout's size: `GetBoundingBox`, which finds the bounds of all shapes in the layout, and `GetTilingBounds`, which returns an explicitly-defined boundary for use when tiling the layout;
3. `MakeVia`, which creates a via shape of the appropriate size for the given layer;
4. `MakePin`, which creates a *pin* shape of the appropriate size for the given layer and a corresponding `Port` object (see Section 2.2.1 for a discussion of pins);
5. `DistributeVias` and `StampVias`, which fill a given span with as many vias as possible given (respectively) the PDK spacing rules or an explicit pitch;
6. methods for drawing wiring either using alternating metal layers with appropriate vias in between (`MakeAlternatingWire`), or by drawing horizontal "fingers" to connect to a larger vertical metal "spine" (`MakeVerticalSpineWithFingers`); and
7. a variety of methods for retrieving shapes that can or can-not participate in routing: `CopyNonConnectableShapesOnLayer`, `CopyConnectableShapesNotOnNets`, and `CopyConnectableShapesOnNets`.

2.3 Geometric Routines

BFG implements a basic set of geometric routines. In fact, it implements the *most basic* set of geometric routines it can. We do not claim that BFG contributes any novelty in its solutions, or even that any of its implementations are optimal in any respect, or even that they are good at all. The field of computational geometry is rich with many pertinent results, and many of its models and assumptions would form a more suitable foundation for such a library. Implementing *optimal* geometric routines was not a goal of this project, but along the way we did become interested in identifying the minimum set of routines necessary to enable such an AMS framework. With this better understood and the necessary software interfaces established, the underlying routines can (and should) be rewritten or deferred to other libraries (Ch. 5).

There are many existing computational geometry libraries, but so far we have not used any. The are four reasons for this. First, each external library comes with its own internal model, and until the BFG layout model was established, any necessary adaptation between the two would have been more hindrance than help. For example, most other libraries tend to use floating point values, whereas BFG enforces integers. Second, most such libraries are complex software packages that would add to our (one-man) maintenance burden. Third,

Table 2.1: Distance functions between two points (x_0, y_0) and (x_1, y_1)

Method	Expression	Name
`Point::L1DistanceTo`	$\lvert x_1 - x_0\rvert + \lvert y_1 - y_0\rvert$	L1 Norm
`Point::L2DistanceTo`	$\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$	L2 Norm
`Point::L2SquaredDistanceTo`	$(x_1 - x_0)^2 + (y_1 - y_0)^2$	L2 Norm Squared

most of the early routines BFG needed were simple and did not justify any of the aforementioned complexity. Fourth, the license for any external library needs to be compatible with BFG's own. Had we found a library that had an adaptable model, was light-weight, and was unrestrictively licensed, we might have skipped this entire process. The most prominent example is the Computational Geometry Algorithms Library [99], which comes with a GPL license that would in turn force BFG to carry a GPL license. Another example, the Boost.Geometry [7] library, is part of the Boost framework of libraries that we have otherwise managed to avoid. Nonetheless, BFG's geometric routines should be replaced with the best examples from literature or through one such library in future work. Clipper2 [45] is a promising new option.

With that out of the way, we start with two notable limitations of BFG's shapes. First, `Rectangle`s must be rectilinear. BFG does **not** support rectangles whose edges are neither parallel nor perpendicular to the line $y = 0$, but these *can* be expressed with a `Polygon`. Second, `Polygons` are simple in the sense that they are assumed to have no self-intersections and no holes. A great deal of computational geometry can make use of this fact. However, BFG does *not* stipulate the order of vertices to be clockwise or counter-clockwise, and so most of those methods do not work. Undoing this limitation would enable the use of many published algorithms for triangulation, intersection, and so on. However, forcing designers to always think of polygons in the same vertex order adds friction to the interface, meaning that the choice isn't without trade-offs.

The two most useful geometric functions in BFG are to calculate 1) the distance between two shapes, and 2) whether two shapes intersect. We discuss these in detail over the next two sections.

2.3.1 Distance

Foundry rules (Sec. 2.2.1) frequently specify minimum and maximum distances between shapes. These include the minimum spacing between shapes on the same layer or on different layers, the minimum enclosure of one shape by another, the maximum distance between a diffusion well and the nearest well tap, and so on. In order to not violate these constraints measuring distance is a necessary and fundamental facility of the framework.

`Points` provide three versions of this, summarised in Table 2.1. The L1 norm `Point::L1DistanceTo` (also known as the Manhattan distance or half-perimeter length) is a computationally efficient proxy for distance. The L2 norm `Point::L2DistanceTo` is the most precise test, but also the most computationally expensive since it requires a square-root. The square of the L2 norm `Point::L2SquaredDistance` allows for comparisons based on Euclidean distance while skipping the square-root.

`Rectangles` provide `Rectangle::ClosestDistanceBetween` (Listing 2.13), which uses the trick of explicitly testing which faces, if any, are parallel (Fig. 2.7).

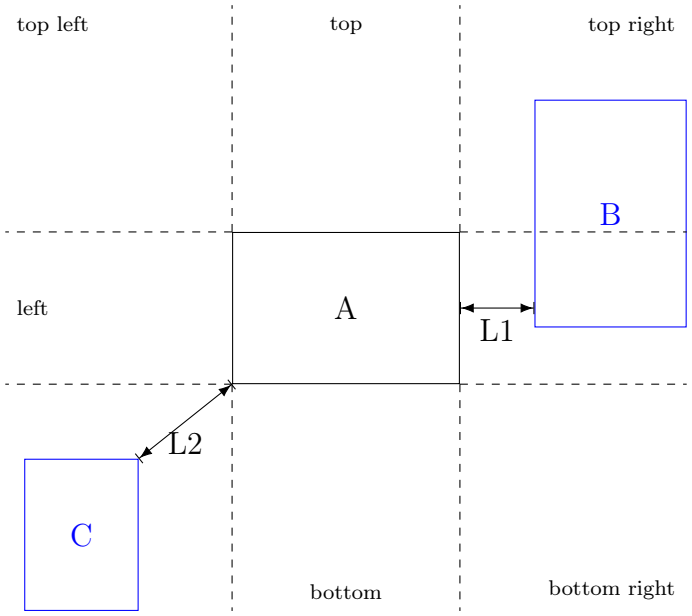


Figure 2.7: Brute force checking for the distance between rectangles

Listing 2.13: Distance between rectangles, from `geometry/rectangle.cc`

```
62 double Rectangle::ClosestDistanceBetween(
63     const Rectangle &lhs, const Rectangle &rhs) {
64   if (lhs.Overlaps(rhs))
65     return 0;

84   bool top = lhs.upper_right().y() < rhs.lower_left().y();
85   bool bottom = lhs.lower_left().y() > rhs.upper_right().y();
86   bool right = lhs.upper_right().x() < rhs.lower_left().x();
87   bool left = lhs.lower_left().x() > rhs.upper_right().x();
88   if (top && right) {
89     return lhs.upper_right().L2DistanceTo(rhs.lower_left());
90   } else if (bottom && right) {
91     return lhs.LowerRight().L2DistanceTo(rhs.UpperLeft());
92   } else if (bottom && left) {
93     return lhs.lower_left().L2DistanceTo(rhs.upper_right());
94   } else if (top && left) {
95     return lhs.UpperLeft().L2DistanceTo(rhs.LowerRight());
96   } else if (right) {
```

Table 2.2: Supported intersections between shapes

	Point	Line	Arc	Rectangle	RoundedRectangle	Polygon
Point	✓	✓	✓	✓		✓
Line		✓	✓	✓		✓
Arc				✓		
Rectangle				✓	✓	✓
RoundedRectangle					✓	✓
Polygon						

```
97      return std::abs(lhs.upper_right().x() - rhs.lower_left().x());
98    } else if (bottom) {
99      return std::abs(lhs.lower_left().y() - rhs.upper_right().y());
100   } else if (left) {
101     return std::abs(lhs.lower_left().x() - rhs.upper_right().x());
102   } else if (top) {
103     return std::abs(lhs.upper_right().y() - rhs.lower_left().y());
104   } else {
105     LOG(FATAL) << "If " << lhs << " and " << rhs << " don't overlap, "
106                << "how did we get here?";
107   }
108   return 0;
109 }
```

`Polygon`s do not implement distance measurements. In fact, most minimum-spacing checks are actually performed by testing if a shape's footprint, when inflated with the minimum separation distances, intersects any other shapes. We will see this next.

2.3.2 Intersections

Several BFG shapes implement an `Overlaps` or an `Intersects` method that tests whether that shape intersects with another. The supported comparison types are summarised in Table 2.2 where, because intersection is a commutative operation, the redundant pairs (in the lower triangle) have been omitted for brevity. Intersections are implemented between: points with other points[2], lines, rectangles, arcs and polygons; lines with other lines, arcs, rectangles and polygons; rectangles with other rectangles, rounded rectangles and polygons; and rounded rectangles with polygons. In general, BFG supports *at least* as many features as are required to produce correct layouts, but only for all layouts it has built so far. What's

[2]Checking for the intersection of two points is checking that they are equal.

more, not all of these features are necessary or even consistent. Intersections between points and arcs were implemented to test for intersections between points are rounded rectangles, but this has not yet been a necessary operation. Polygon-on-polygon intersection has not been necessary for BFG's designs so far (Ch. 4), but is generally desirable and so should be implemented in later work (a stand-in function currently tests for the intersection of their bounding boxes before giving up). Additionally, intersections with arcs were initially intended to facilitate intersection checks between two rounded rectangles but, as we will see, it was more straightforward to implement the check without them.

Points

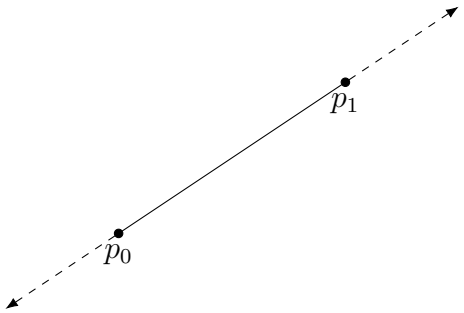


Figure 2.8: BFG `Lines` are both bounded and unbounded, depending on who's asking

BFG `Lines` represent a straight line between two points p_0 and p_1, but they can also represent the infinite line running through those two points (see Figure 2.8). Accordingly, intersection with a point comes in two flavours: `Line::Intersects` and `Line::IntersectsInBounds`. The former tests if the the point is on the infinite line, and the latter tests if it is between p_0 and p_1. Because `Line` objects are usually created by specifying these two points, it is most convenient for us to compute the gradient m and y-offset C to represent the line as $y = mx + C$. Vertical lines are treated as a special case. That in turn means that most convenient way for us to check if a point $p_2 = (x_2, y_2)$ lands on the line is to test the error in y, Δy:

$$\hat{y} = mx_2 + C$$
$$\Delta y = |\hat{y} - y_2|$$

If Δy is strictly less than 1, the minimum spacing of our coordinate space, we consider it an intersection. The `Line::IntersectsInBounds` variant then simply checks if the given point lies within the bounding points:

$$\min(x_0, x_1) \leq x_2 \leq \max(x_0, x_1) \text{ and } \min(y_0, y_1) \leq y_2 \leq \max(y_0, y_1)$$

This is, incidentally, the same test as for the intersection of a point with a rectangle defined by opposite corners p_0 and p_1. Since a `Rectangle` enforces the invariant that the lower left coordinates are less than or equal to the upper-right ones, we set p_0 to the lower-left coordinate and p_1 to the upper-right one, and the test simplifies to:

$$x_0 \leq x_2 \leq x_1 \text{ and } y_0 \leq y_2 \leq y_1$$

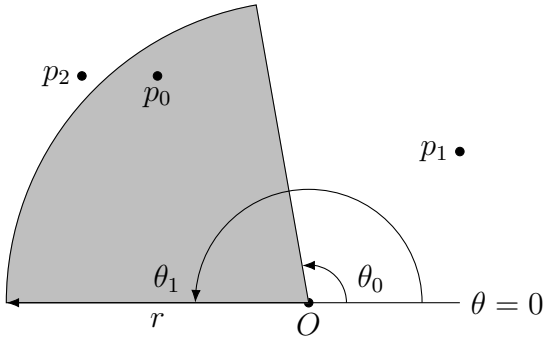
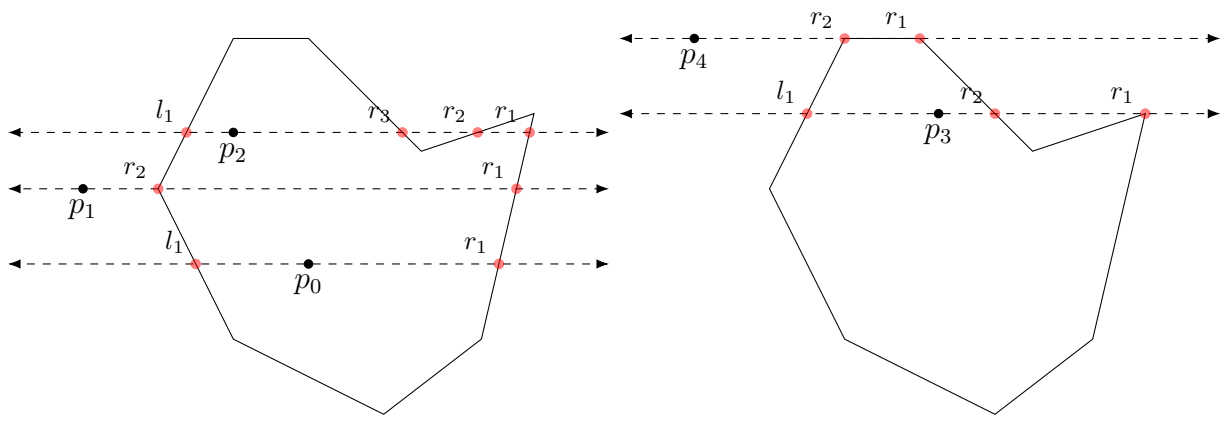


Figure 2.9: BFG `Arcs` and whether or not they intersect points

The test for whether a point falls within an arc is also trivial. Per Figure 2.9, BFG's `Arc` describes a typical geometric arc of radius r from angle θ_0 through angle θ_1 about origin O. A point p is considered within the arc if its angle to the origin θ_p satisfies $\theta_0 \leq \theta_p \leq \theta_1$ *and* distance to the origin is less than the arc's radius, or $||p - O||_2 \leq r$. In the example, p_1 is within the radius but outside of the angular range; p_2 is within the angular range but outside of the radius; and p_0 is just right.

Testing whether a point intersects a *polygon* is typically performed by first triangulating the polygon, then testing whether any of the constituent triangles contain the point through a barycentric coordinate system, system of parametric equations or, famously, by checking on which side of each edge the point lies with the dot product. These would all be better ways, but because BFG's polygons do not enforce a vertex order, triangulation is not trivial. Instead, BFG implements the even-odd rule [29]: it casts two opposite pseudo-infinite rays (`Lines`) outward and then tests how many times that ray intersects with the polygon on each side of the test point. The ray angle is arbitrary, so is chosen to be 0. Conceptually, the rays are infinite, but in reality they only need to extend just past the bounding box of the polygon. Each intersection of a ray with an edge of the polygon is an entry or an exit; if the numbers of entries and exits do not match by the time we reach "infinity", we must have started inside the polygon.

Consider the example of Figure 2.10a, where three points p_0, p_1 and p_2 are being tested for intersection with a polygon. Both p_0 and p_2 have an odd number of intersections to their left and right, so we know that they are in the polygon; p_1 has 0 intersections to the left (even) and 2 to the right (also even), so we know it falls outside the polygon. The



(a) Nominal cases (b) Annoying cases

Figure 2.10: Casting a ray to test point-polygon intersection

disadvantage of this method is its tricky edges cases: consider now the example of Figure 2.10b, where we again cast rays to test the intersection of points p_3 and p_4 with the polygon. This time, point p_3 counts two intersections to the right and only one to the left. At first this contradiction is frightening, but the resolution is simple. Intersections with polygon vertices *must* count as both an entry *and* and exit, an invariant that must be enforced by our routine for the intersections of lines and polygons. The problem with point p_4 is that its ray is *incident* on one of the polygon edge lines. Because we consider a shape's edge to be part of the shape, this also *must* count as an entry *and* an exit. As we shall see, enforcing this is trickier than it seems.

Lines

The intersection of a line and a rectangle, as with a line and a polygon, is simply the union of the intersections of the line with each line that defines the rectangle's (or polygon's) boundary. The intersection of a line $y_1 = m_1 x + c_1$ with another line $y_2 = m_2 x + c_2$ is the familiar and straightforward solution to setting their equations equal to one another, then find the intersecting point (x_i, y_i).

$$\begin{aligned} m_1 x_i + c_1 &= m_2 x_i + c_2 \\ x_i(m_1 - m_2) &= c_2 - c_1 \\ x_i &= \frac{c_2 - c_1}{m_1 - m_2} \\ y_i &= m_1 x_i + c_1 \end{aligned}$$

The usual exceptions to this are when either line is vertical (m undefined) or if they are parallel/anti-parallel ($m_1 - m_2 = 0$), in which cases the algebra is adjusted accordingly.

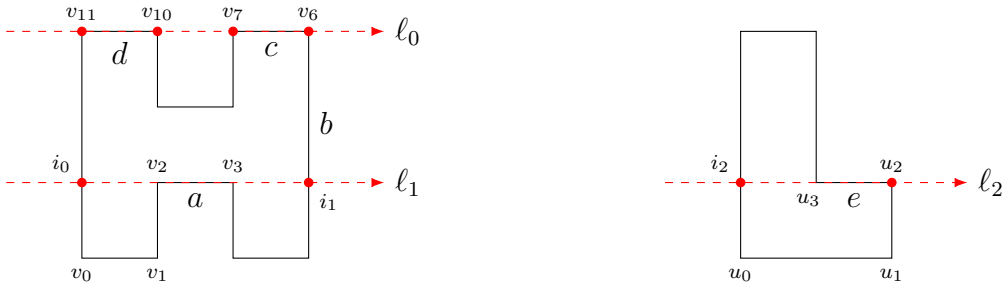


Figure 2.11: Coincidence of lines with the edges of non-convex polygons

There are only three notable aspects of these routines. First, the `Line` class provides tests for whether neither line, the first line, the second line, or both lines are to be considered infinite. Second, the caller is returned both the intersection point, if any, and whether the lines were coincident. Third, the caller can opt to exclude the start and end point of the line in a bounded intersection, for use in the rectangle and polygon cases we will discuss next.

The intersection of a line and a rectangle starts by forming the four boundary edges (north N, east E, south S, west W) of the rectangle from the lower-left (x_0, y_0) and upper-right (x_1, y_1) coordinates:

$$\begin{aligned}
\ell_W &: (x_0, y_0) \to (x_0, y_1) \\
\ell_N &: (x_0, y_1) \to (x_1, y_1) \\
\ell_E &: (x_1, y_1) \to (x_1, y_0) \\
\ell_S &: (x_1, y_0) \to (x_0, y_0)
\end{aligned}$$

It then proceeds as expected, testing each boundary line for intersection, then collecting the results for the caller. However, there is an interesting case when the given line is coincident with an edge, as is common in rectilinear layout. In such a case the tested line will intersect with three boundary edges: at the end of one, coincident with the second, and at the start of the third. To account for this, boundary lines are tested for intersection with their start point excluded (they are open at one end). To enforce entry/exit parity for use in the even-odd rule, care is then taken to duplicate intersections when a line strikes a single vertex.

We generalise this approach to the intersection of polygons with lines. BFG `Polygons` are defined by a list of vertices (`Points`) $(v_0, v_1, \ldots, v_N)$, from which we form the boundary lines:

$$\mathbb{L} = ((v_0, v_1), (v_1, v_2), \ldots, (v_{N-1}, v_N), (v_N, v_0))$$

Except that we must adjust the algorithm from the `Rectangle` case. Rectangles are convex, so any line incident on a boundary line is guaranteed to be entering and then exiting the shape. This is not true for the simple polygons that BFG supports. Consider the three lines l_0, l_1 and l_2 in Figure 2.11, which each coincide with at least one edge of the example polygons. Line l_0 matches the simplest case because it lands on the two external faces d and

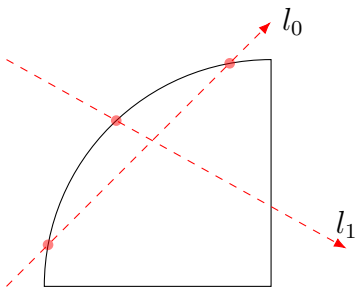


Figure 2.12: A line intersecting an arc

c. We would expect the line to (enter, exit) at the vertices (v_{11}, v_{10}) and (v_7, v_6). However, line l_1 coincides with a polygon edge *while* travelling within the polygon, and so should *not* record an entry or exit there. We expect l_1 to (enter, exit) at (i_0, i_1). Line l_2 requires a still different treatment in its coincidence with edge e: since the line enters the polygon at i_2, only the exit at u_2 should be recorded. If l_2 were reversed, the first intersection with e ought to be recorded as an entry, and i_2 as an exit.

Clearly, the treatment of a coinciding edge depends on the intersections of the line with the polygon before and after the edge. We resolve this problem by first collecting as much information about all intersecting points as possible and then deciding one-by-one whether a point or edge represents an entry or an exit. To facilitate this, a `geometry::PointOrChoice` class is introduced to explicitly track the ambiguities at each such edge. Once collected, the intersecting points and edges are sorted by their position along the incident line in order to remove dependence on the line's direction. To make this easier, `PointOrChoice` provides methods to determine the nearest or furthest end of an edge from a reference point.

On the other hand, the intersection of lines and *arcs* (Fig. 2.12) is comparatively straightforward. BFG's `Arcs` exist only in service of creating `RoundedRectangle`, so the intersection with a line only tests for intersection with the *outer arc*, not the radius. This reduces to setting the equation of the line equal to that of the circle and solving the resulting quadratic. The last step is to ensure that the intersecting points fall within the angular range of the arc. The elementary algebra at work here is perhaps less interesting to the reader than an example of how BFG implements these routines. Listing 2.14 reproduces this method below. Note that, as with most `Line`-related code, an additional function exists to check if the intersection occurs within the `Line`'s bounds.

Listing 2.14: Intersecting a line with an arc, from `geometry/arc.cc`

```
48 std::vector<Point> Arc::IntersectingPoints(const Line &line) const {
49   // Equation of circle is
50   //
51   //  (x - x_c)^2 + (y - y_c)^2 = r^2
52   //
53   // where x_c, y_c = centre_ and r = radius_.
54   double x_c = centre_.x();
55   double y_c = centre_.y();
```

```
56    double r = radius_;
57
58    // If vertical, we substitute x = k into the circle equation:
59    //
60    //               (k - x_c)^2 + (y - y_c)^2 = r^2
61    //                    y^2 - 2y_c*y + y_c^2 = r^2 - (k - x_c)^2
62    // y^2 - 2y_c*y + y_c^2 - r^2 + (k - x_c)^2 = 0
63    //
64    // which is quadratic
65    //                            Ay^2 + By + C = 0
66    //
67    // with A = 1
68    //      B = -2y_c
69    //      C = y_c^2 - r^2 + (k - x_c)^2
70    //
71    // so we just solve the quadratic formula:
72    //      y = (-B +- sqrt(B^2 - 4AC)) / 2A
73    if (line.IsVertical()) {
74      double k = line.start().x();
75      double B = -2.0 * y_c;
76      double C = y_c * y_c - r * r + (k - x_c) * (k - x_c);
77      std::vector<double> y_values = Utility::SolveQuadraticReal(1.0, B, C);
78
79      std::vector<Point> result;
80      for (double y_value : y_values) {
81        Point point(k, y_value);
82        if (IsPointInArcBounds(point)) {
83          result.push_back(point);
84        }
85      }
86      return result;
87    }
88
89    double m = line.Gradient();
90    // Little c! The line's offset at the y-intercept.
91    double c = line.Offset();
92
93    // If not vertical, equation of line is y = mx + c.
94    //
95    // Substituting y into the circle equation,
96    //
97    //                                (x - x_c)^2 + (mx + (c - y_c))^2 = r^2
98    //     x^2 -2x_c*x + x_c^2 + m^2*x^2 + 2m(c - y_c)*x + (c - y_c)^2 = r^2
99    // x^2 -2x_c*x + x_c^2 m^2*x^2 + 2m(c - y_c)*x + (c - y_c)^2 - r^2 = 0
100   // (1 + m^2)x + (2m(c - y_c) - 2x_c)*x + x_c^2 + (c - y_c)^2 - r^2 = 0
101   //
102   // This is a quadratic in x,
103   //                           Ax^2 + Bx + C = 0
104   //
105   // with A =  (1 + m^2)
106   //      B = 2(mc - my_c - x_c)
107   //      C = x_c^2 + (c - y_c)^2 - r^2
108   double A = 1 + m*m;
109   double B = 2*(m*(c - y_c) - x_c);
110   double C = x_c * x_c + (c - y_c) * (c - y_c) - r * r;
111   std::vector<double> x_values = Utility::SolveQuadraticReal(A, B, C);
112
113   std::vector<Point> result;
114   for (double x_value : x_values) {
115     Point point(x_value, m*x_value + c);
116     if (IsPointInArcBounds(point)) {
117       result.push_back(point);
118     }
```

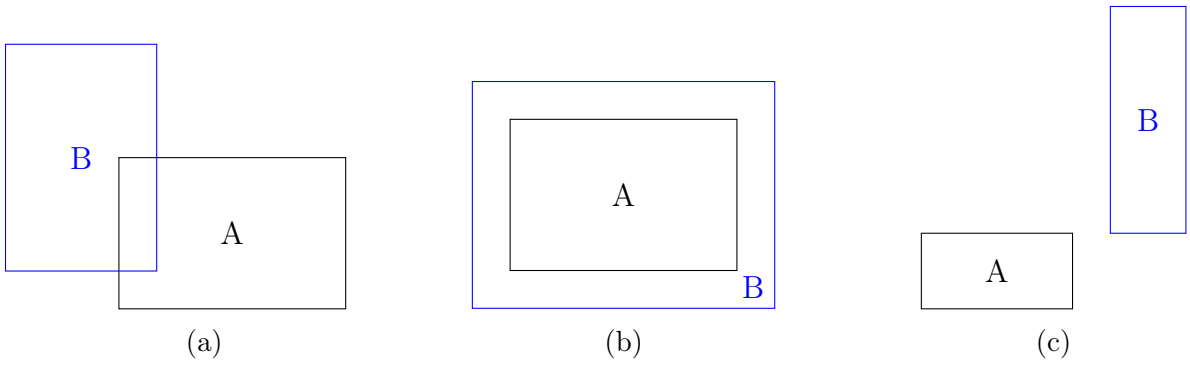


Figure 2.13: Intersections between rectangles

```
119   }
120   return result;
121 }
```

Rectangles

To test if a rectangle A intersects with another rectangle B, we simply test each possible case: 1) the corner point of one rectangle intersects the other (Fig. 2.13a) (using the rectangle-point intersection operation described above); 2) the first rectangle is entirely contained within the second (Fig. 2.13b), or vice versa; and 3) there is no intersection (Fig. 2.13c).

Polygons

To determine whether a rectangle and a polygon intersect, BFG relies on both 1) the odd-even rule we established for detecting points within a polygon; and 2) the line-polygon intersections we have discussed. Because our polygons are simple, not convex, there are broadly five cases to consider:

1. (Fig. 2.14a.) If the bounding boxes of a rectangle A and polygon B do not overlap, there is no intersection.
2. (Fig. 2.14b.) At least one vertex of rectangle A lands inside and one vertex lands outside of polygon B.
3. (Fig. 2.14c.) Rectangle A is contained entirely within polygon B.
4. (Fig. 2.14d.) The polygon B is contained entirely within the rectangle A.
5. (Figs. 2.14e and 2.14f.) The polygon B surrounds the rectangle A, so their bounding boxes overlap, but they themselves do not.

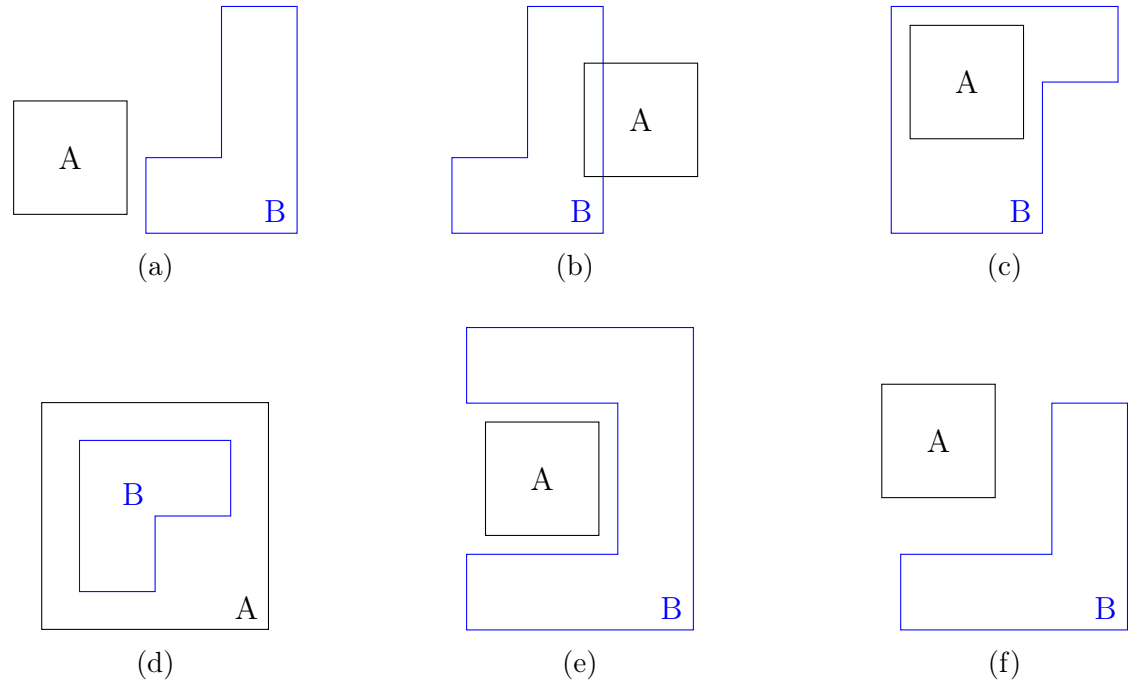


Figure 2.14: Intersections between rectangles and polygons

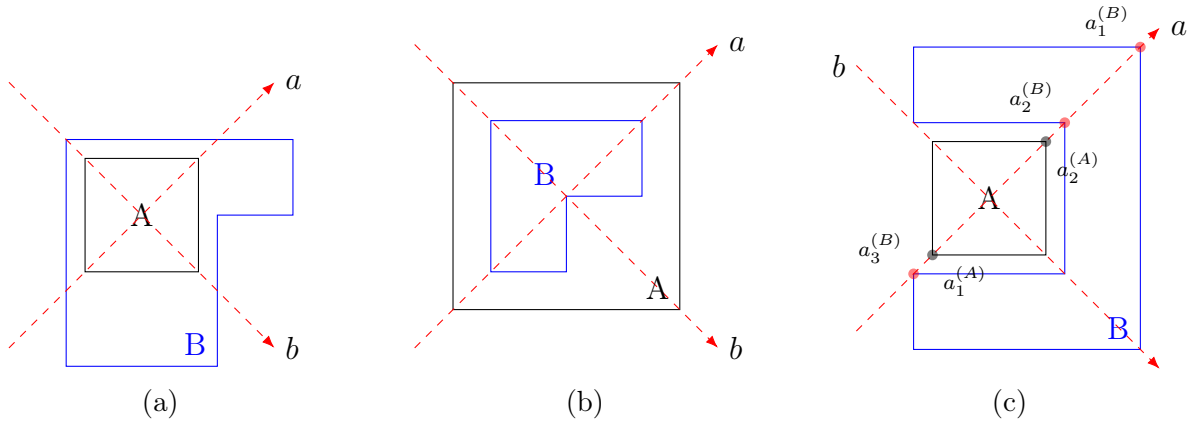


Figure 2.15: Ray-casting to find the intersection between rectangles and polygons

Because it is both computationally inexpensive to test bounding boxes (they are just rectangles) and most pairs of shapes in a design do *not* overlap, BFG first tests whether the bounding boxes of the polygon and the rectangle collide. (The bounding box of the rectangle is just the rectangle itself.) If they do, we move onto the next most common case (2): every boundary line of the rectangle is tested for intersection with every boundary line of the polygon as discussed above. Still, if no intersections are found, we test for the three

cases (3), (4), (5) with a variation on the even-odd rule. Figures 2.15a - 2.15c illustrate this process for the examples from Figures 2.14c - 2.14e, respectively. The diagonals of the rectangle are extended to pseudo-infinite rays a and b, which are then intersected with the polygon. Because our polygon-line intersection returns pairs of (entry, exit) points on the polygon, it is then sufficient to test if any (entry, exit) span coincides with the inside of the rectangle.

Take Figure 2.15c in particular. Intersection of the diagonal a with the polygon B should result in the (entry, exit) point pairs $(a_3^{(B)}, a_3^{(B)})$ and $(a_2^{(B)}, a_1^{(B)})$. We also know that the pair $(a_1^{(A)}, a_2^{(A)})$ defines the entry into and exit out of the rectangle. We can reduce these points to their scalar distance along the line a with a familiar result from linear algebra.

Let p_0 and p_1 be two points that define the line a, such as its start and end. Then, let $\vec{a}$ be the vector from p_0 to p_1. The projection of point g onto line a is given by

$$\begin{aligned}\text{proj}_{\vec{a}}\left(\vec{g}\right) &= \frac{\vec{g}\cdot\vec{a}}{\vec{a}\cdot\vec{a}}\vec{a} \\ &= \frac{\vec{g}\cdot\vec{a}}{||\vec{a}||^2}\vec{a}\end{aligned}$$

where $\vec{g}$ is defined as the vector from point p_0 to point g. Because we are only interested in a scalar representation of the relative position of each projection, we need only calculate

$$\text{coeff}_{\vec{a}}\left(\vec{g}\right) = \frac{\vec{g}\cdot\vec{a}}{||\vec{a}||^2}$$

BFG's `Line` class provides a `Line::ProjectionCoefficient` method that will form the appropriate vector out of the given point and then compute this coefficient. Denote this as

$$R : g \mapsto \text{coeff}_{\vec{a}}\left(\vec{g}\right)$$

The only thing left to do is to find the projection coefficient of each point in the (entry, exit) intersection point pairs and check if any of them overlap. In this case, we end up with two pairs of scalar spans

$$\left[R\left(a_3^{(B)}\right), R\left(a_3^{(B)}\right)\right], \left[R\left(a_2^{(B)}\right), R\left(a_1^{(B)}\right)\right]$$

that we need to check for overlap with the scalar span of the rectangle itself:

$$\left[R\left(a_1^{(A)}\right), R\left(a_2^{(A)}\right)\right]$$

The corresponding code snippet is reproduced in Listing 2.15.

Listing 2.15: Casting rays to detect if a polygon contains or surrounds a rectangle, from `geometry/polygon.cc`

```
555  std::vector<Line> test_lines = {
```

```
556       Line(rectangle.lower_left(), rectangle.upper_right()),  // diagonal
557       Line(rectangle.UpperLeft(), rectangle.LowerRight())     // other diagonal
558   };
559
560   for (const auto &test : test_lines) {
561     // The projection back onto diagonal is measurable as a scalar coefficient
562     // multiplied by the original vector diagonal defines. As shorthand we just
563     // call these the 'coefficients'.
564     double start_coefficient = 0.0;  // Distance from start to start along the
565                                      // line is always zero.
566     double end_coefficient = test.ProjectionCoefficient(test.end());
567
568     std::vector<PointPair> points = IntersectingPoints(test);
569
570     for (const auto &point_pair : points) {
571       const Point &entry = point_pair.first;
572       const Point &exit = point_pair.second;
573
574       double entry_coefficient = test.ProjectionCoefficient(entry);
575       double exit_coefficient = test.ProjectionCoefficient(exit);
576
577       if (entry_coefficient <= start_coefficient &&
578           exit_coefficient >= start_coefficient) {
579         return true;
580       }
581       if (entry_coefficient <= end_coefficient &&
582           exit_coefficient >= end_coefficient) {
583         return true;
584       }
585     }
586   }
```

We repeat this using the opposite diagonal b for completeness. If only one diagonal is checked, it's possible to miss a polygon that intersects the bounding box but not the rectangle itself, as in the example of Figure 2.14f. It should still be correct, however, since we will only apply this test if the polygon's boundaries *do not* intersect with the rectangle's boundaries *and* if the bounding boxes overlap.

An alternative to this approach would be use the odd-even rule and simply test whether any vertex of the rectangle lands inside the polygon. A special case could then be made for case (4), where we test whether the polygon bounding box fits entirely within the rectangle. I[3] did not do this because it felt more complicated, though in reality the additional complexity is deferred to other routines. Additionally, it is strongly desirable to have formal proof that these methods work, but we leave that to future work.

Rounded Rectangles

A rounded rectangle of length l, width w and corner radius r can be decomposed into 9 regions as illustrated in Figure 2.16. There are five rectangular regions $A - E$ and four arc regions $F - I$. They are useful for representing the minimum area around a rectangular shape, such as a via or metal pour, that must exclude other shapes. Consequently, we are

[3] I break from the royal "we" here on purpose.

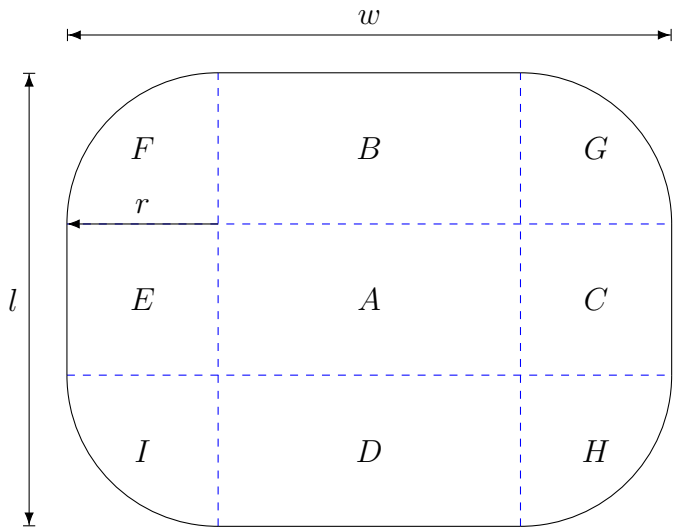
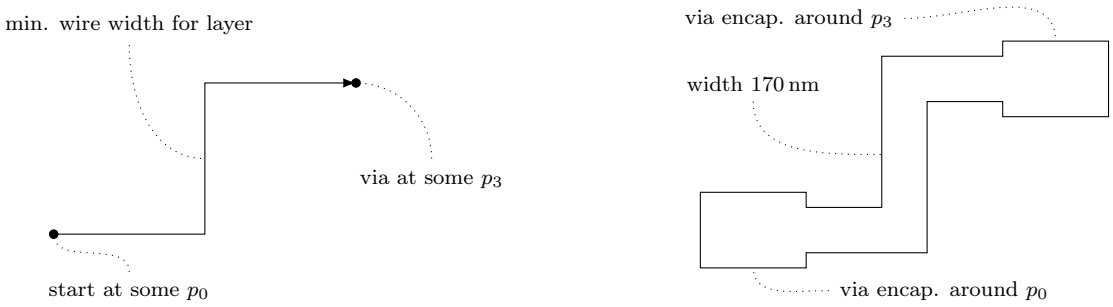


Figure 2.16: A rounded rectangle and its regions

mostly interested in whether a rounded rectangle overlaps with other rectangles or with polygons. Unfortunately, the latter is incomplete, pending implementation of a polygon-arc intersection test based on the principles and functions set out above.

Testing for the intersection of a rounded rectangle P with a regular rectangle Q is well supported. The algorithm is greatly simplified through the successive elimination of the quadrants in which Q might appear, relative to P. First, we test for the overlap of their bounding boxes, an optimisation we have seen frequently. Second, we test for the intersection of Q with each of the P's inner rectangle regions, $A - E$. If Q does not intersect with any of these, then because we have already established that their bounding boxes do overlap, it must intersect with one of the corner regions $F - I$. Additionally, because these rectangles are rectilinear, any overlap in those regions must correspond to the opposite diagonal point on Q: in the top left (F), any overlap would start with the bottom right corner of Q; in the top right (G), any overlap must start with the bottom left corner of Q; and so on. Therefore, it is sufficient to check if the distance from the centre of each corner arc to the corresponding point on Q falls within the arc's radius.

The algorithm for determining whether a rounded rectangle P intersects with *another* rounded rectangle R is similar. Because rounded rectangles must have rectilinear central regions we can greatly simplify this procedure as well. We start with the usual bounding-box check. Then, we test if any of the central regions $A - E$ of P and Q overlap with one another. If not, the overlaps must mutually occur in the corner regions $F - I$. This is a result of the x- and y-axis alignment that BFG enforces on rectangles (and not generally true of a rectangle that can be arbitrarily rotated). We again test for the distance between rounded rectangle's diagonally-opposed corner regions. Unlike in the case of P and Q above, this time we must check the distance between the centres of the corner arcs. There is a



(a) Abstract wire path (b) Resulting polygon

Figure 2.17: Optimistic cartoon of a wire path and its resulting polygon



(a) (b)

Figure 2.18: Examples of notches between non-intersecting wire edges

collision between P and R if the distance between them is less than or equal to the sum of their corner radii.

2.3.3 Wire Paths

A common need when laying out digital circuits is to draw a minimum-sized wire along some path. These paths typically encounter vias that connect them to structures on other layers, in which case the path must be resized to correctly encapsulate the via with metal (Fig. 2.4). Though the designer could methodically calculate an appropriate polygon to represent the wire each time, that would be tedious. It is preferable to think of a wire as tracing the line through its center, from the center of some via, turning at some corners, and arriving at some other via, as in Figure 2.17a. The abstract blueprint of the wire should then be turned into a correctly-sized polygon, as in Figure 2.17b.

In BFG, wire paths are paths implemented by the `PolyLine` class. `PolyLine` objects store a starting point p_0 and list of `LineSegments`, which pair successive points in the line p_i with the nominal width of that segment w_i. At any time, the `PolyLine` can be "inflated" into

a `Polygon` using the `PolyLineInflator`[4]. Additional fields determine how far before the first and after the last point the inflated line should extend. `PolyLine` allows the user to specify the location of a via through the `PolyLine::InsertBulge` method, which will modify the line segments around that point to ensure that the resulting polygon is at least as wide and long as needed.

What sets BFG's `PolyLine` apart as an abstract wire, over what is commonly called a "PolyLine" in other geometry libraries, is that it can enforce layout-related invariants that are particular to wires. The first of these is that wires must be *rectilinear*, meaning that any two successive points $p_i = (x_i, y_i)$ and $p_{i+1} = (x_{i+1}, y_{i+1})$ must share an x- or y-coordinate:

$$x_i = x_{i+1} \text{ or } y_i = y_{i+1}$$

If both are true, the two successive points are equal and therefore one is redundant. The `PolyLine` will set the width of that segment to the largest of the two and remove one of the duplicate end-points. It will also remove successive segments in a straight line if they have the same width, because this could be expressed more simply with a single segment. The rectilinear requirement can be relaxed if appropriate for the design (or, more likely, the PDK).

Wherever two non-intersecting edges of the final wire polygon would violate the minimum separation of the layout layer, a "notch" is formed. See Figure 2.18 for examples. If its `min_separation_` field is set, `PolyLine` attempts to find and remove segments that would cause notches. This is handled according to where the notch is encountered: notches in a straight line (Fig. 2.18b) can be removed by widening the notch segment to the minimum width of its two neighbours; notch segments around a corner must be widened to close any gap. The reader is referred to the `PolyLine::EnforceInvariants` method in `geometry/poly_line.cc`, which implements these fixes with the predictably named methods:

```
void ReplaceDuplicateEndPointsWithWidest();
void RemoveRedundantSegments();
void RemoveNotchesInAStraightLine();
void RemoveNotchesAroundCorners();
```

When finalising a layout, a `PolyLineInflator` must turn the specifications of the `PolyLine` into the corresponding polygon that represents the wire. To illustrate the procedure, consider the detailed view of an example 3-segment path modeled by `PolyLine` in Figure 2.19. We take into account:

- the starting point, p_0;
- each of segments $s_i = (p_i, w_i) \mid i \in \{1, \ldots, N\}$, which define the end point p_i and the width of the wire w_i therein; and

[4]The separation of `PolyLineInflator` and `PolyLine` stems from an original view of `PolyLine` as technology-independent. The `PolyLineInflator`, which must be aware of physical properties in order to correctly size the `PolyLine`, factors out the physical concern from `PolyLine` itself. This was a premature optimisation; I would not insist that it be done this way again.

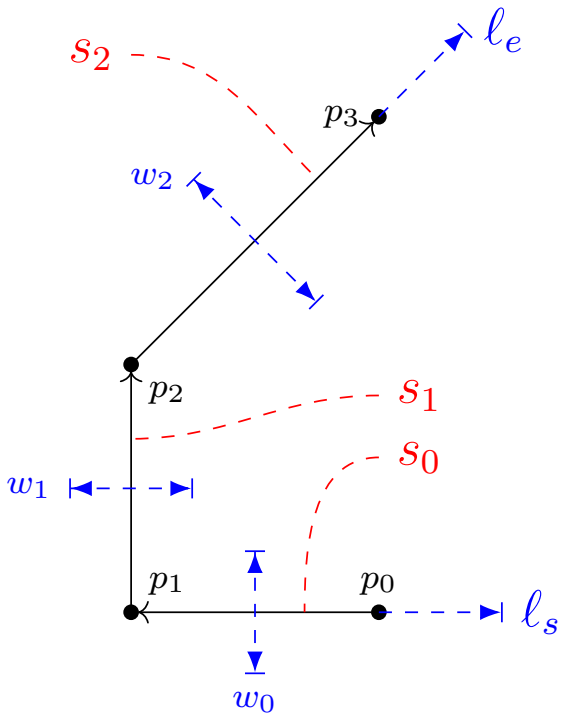


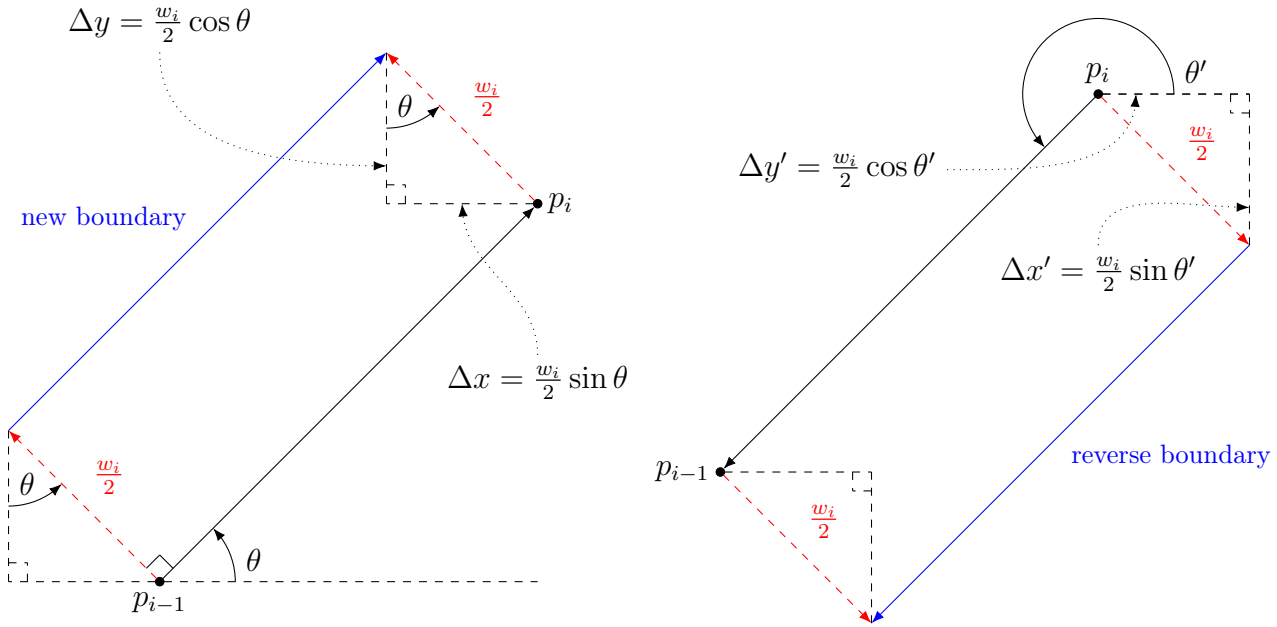
Figure 2.19: An example 3-segment wire modeled by a `PolyLine`

- the extensions to the wire length at the start ℓ_s and end ℓ_e of the line.

We do not assume that paths are rectilinear for this process in order that the class may be more generally useful, though assuming as much could yield considerable simplification if the procedure needs to be optimised.

The algorithm starts by generating the boundary lines for each segment in the forward and reverse directions. In the forward direction, boundary lines are generated for each segment $s_i = (p_i, w_i)$ by shifting the central line (p_{i-1}, p_i) out by a distance of $w_i/2$ perpendicular to the line (Fig. 2.20a). The boundary line for the starting segment is extended back from the nominal starting point by ℓ_s if $\ell_s \neq 0$; the boundary line for the final segment s_N is extended past the nominal end point by ℓ_e if $\ell_e \neq 0$; and the boundary lines for middle segments are drawn to match the lengths of their generating segments. The process is repeated in reverses, iterating backwards through the list of segments *and reversing each line* (Fig. 2.20b). That is, we then generate boundary lines for the lines $(p_0, p_1), (p_1, p_2), \ldots, (p_{N-1}, p_N)$ and then for $(p_N, p_{N-1}), (p_{N-1}, p_{N-2}), \ldots, (p_1, p_0)$.

To generate a particular boundary line (b_{i-1}, b_i) for segment i, we use the trigonometric



(a) Forward line (b) Reverse line

Figure 2.20: Generated boundary lines for segments of a `PolyLine`

relationships shown in Figure 2.20 to offset the central line (p_{i-1}, p_i) according to

$$b_i = p_i + (\Delta x, \Delta y)$$
$$b_{i-1} = p_{i-1} + (\Delta x, \Delta y)$$

where

$$\Delta x = \frac{w_i}{2} \sin\theta$$
$$\Delta y = \frac{w_i}{2} \cos\theta$$

In the reverse direction we create the boundary line (b'_i, b'_{i-1}) from the reversed central line (p_i, p_{i-1})

$$b'_i = p_i + (\Delta x', \Delta y')$$
$$b'_{i-1} = p_{i-1} + (\Delta x', \Delta y')$$

where

$$\Delta x' = \frac{w_i}{2} \sin\theta'$$
$$\Delta y' = \frac{w_i}{2} \cos\theta'$$

The angle that the reversed line forms to the horizon, θ', is offset from that of the forward direction by π:

$$\theta' = \theta + \pi$$

From the trigonometric identities $\sin(x+\pi) = -\sin x$ and $\cos(x+ß) = -\cos x$, reversing the central line has the intended effect of finding the opposite offsets:

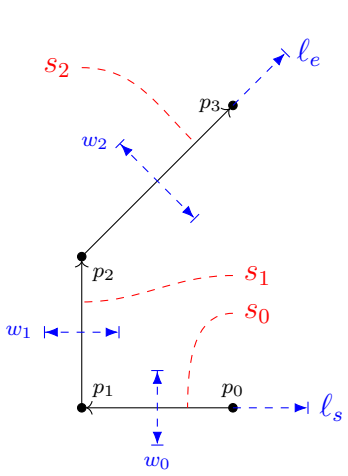
$$\begin{aligned} \Delta x' &= \frac{w_i}{2} \sin\theta' & \Delta y' &= \frac{w_i}{2} \cos\theta' \\ &= \frac{w_i}{2} \sin(\theta+\pi) & &= \frac{w_i}{2} \cos(\theta+\pi) \\ &= -\frac{w_i}{2} \sin(\theta) & &= -\frac{w_i}{2} \cos(\theta) \\ &= -\Delta x & &= -\Delta y \end{aligned}$$

BFG's `Line` class implements this procedure in a convenient `Line::Shifted` instance method. A line can return an appropriately shifted copy of itself with optional extensions to the start and end points.

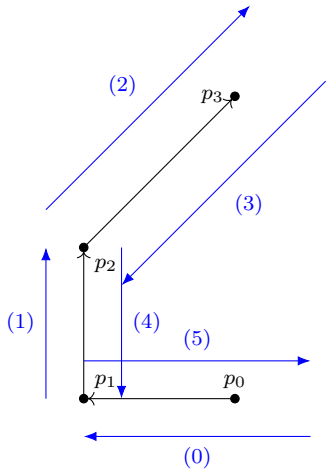
Listing 2.16: The `Line::Shifted` instance method will generate a shifted version of a line, from `geometry/line.cc`

```
118 Line Line::Shifted(const Line &source,
119                     double width,
120                     double extension_source,
121                     double extension_end) {
122   // TODO(aryap): Integer division can lead to precision loss here,
123   // so we *should* make sure we recover it.
124   double theta = source.AngleToHorizon();
125
126   int64_t shift_x = static_cast<int64_t>(std::sin(theta) * width);
127   int64_t shift_y = static_cast<int64_t>(std::cos(theta) * width);
128
129   Line shifted_line(source);
130   shifted_line.Shift(-shift_x, shift_y);
131
132   if (extension_source > 0.0) {
133     int64_t extension_x =
134         static_cast<int64_t>(std::cos(theta) * extension_source);
135     int64_t extension_y =
136         static_cast<int64_t>(std::sin(theta) * extension_source);
137     shifted_line.ShiftStart(extension_x, extension_y);
138   }
139
```

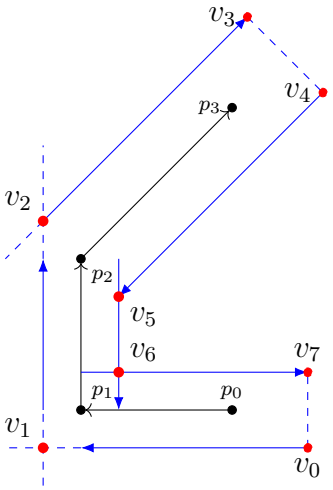
```
140   if (extension_end > 0.0) {
141     int64_t extension_x =
142         static_cast<int64_t>(std::cos(theta) * extension_end);
143     int64_t extension_y =
144         static_cast<int64_t>(std::sin(theta) * extension_end);
145     shifted_line.ShiftEnd(extension_x, extension_y);
146   }
147
148   return shifted_line;
149 }
```



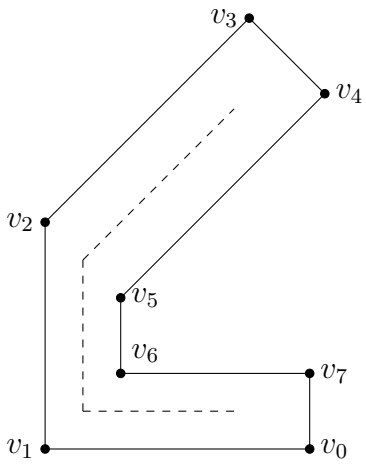
(a) Figure 2.19 again



(b) Boundary lines formed



(c) Boundary intersections found



(d) Polygon formed

Figure 2.21: Procedure for inflating a `PolyLine` into a `Polygon`

The overall process is summarised in Figure 2.21. Once the forward boundary lines

$$\mathcal{F} = \{(b_0, b_1), (b_1, b_2), \ldots, (b_{N-1}, b_N)\}$$

and the reverse boundary lines

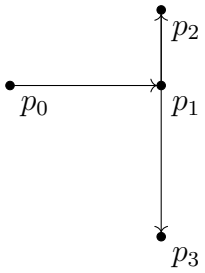
$$\mathcal{R} = \left\{(b'_N, b'_{N-1}), (b'_{N-1}, b'_{N-2}), \ldots, (b'_1, b'_0)\right\}$$

are assembled (Fig. 2.21b), we iterate through the combined list $\mathcal{B} = \mathcal{F} \| \mathcal{R}$ and intersect each successive pair of lines (Fig. 2.21c). We treat the intermediate boundary lines as infinite when finding intersections because they will have to be extended to meet further boundary lines when, for example, turning a corner. Where there is no intersection in intermediate parallel segments, the two boundary lines are joined by a perpendicular line at their finite limits — this is also how the forward and reverse boundaries are joined. Next, these intersections are taken to form the vertices of the final polygon (Fig. 2.21d). At the start (and end) of the list we only need the start (and end) point of the boundary line.

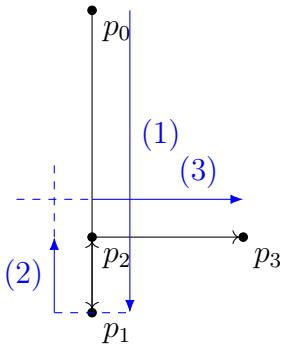
Unfortunately, there are several complications that arise when trying to accommodate the most general wire paths. Notable examples are shown in Figures 2.22. First, even if wires are constrained to be rectilinear, allowing them to reverse direction can lead to odd structures (Fig. 2.22a). This isn't always a problem, but in some orientations the boundary lines can create a loop (Fig. 2.22b). To detect this, we check every new boundary line for intersection (within bounds) with *every other* boundary line beyond the immediate neighbour. In the figure, line (1) intersecting with line (3) is a detectable anomaly. This also demonstrates why redundant segments should be removed: it is easier to preclude the possibility of multiple consecutive about-faces than to deal with the boundary lines in such a scenario.

We also take particular care to remove notches. Because these can form across multiple short and narrow segments, we must check for notches between each segment and every successive segment that is smaller or equal in width (Fig. 2.22c). When turning corners, we must check the distance to the intersection with following segment (Fig. 2.22d) lest a notch form there too (Fig. 2.22e).

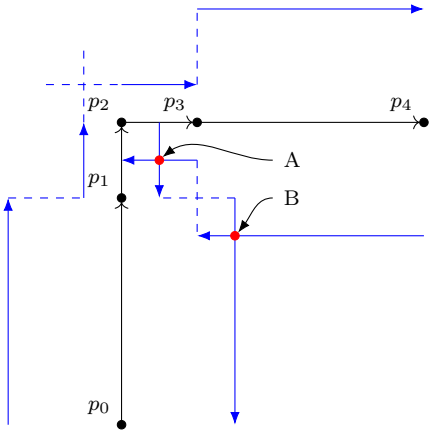
BFG's wire path is inspired by the "stick diagrams" that were once popular for manually sketching IC layouts in abstract terms (see [110], pp. 28-29). We wondered if all of BFG's layouts could be specified this way; others have also seen merit in re-popularising this idea recently [73]. Additionally, BFG includes `AbstractVia` and a whole `PolyLineCell` class, both designed to keep topological information about the circuit layout free of physical details until the last minute. We have found that there is significant complexity in doing this without a set of simplifying invariants (restrictions), but that it is also very convenient. Put another way, the designer must be willing to accept significantly less control for simplicity. While BFG's `PolyLine` is not bug-free, it nonetheless presents the designer with a tenable balance between ease-of-use and correctness. If used correctly, the abstractions hold well.



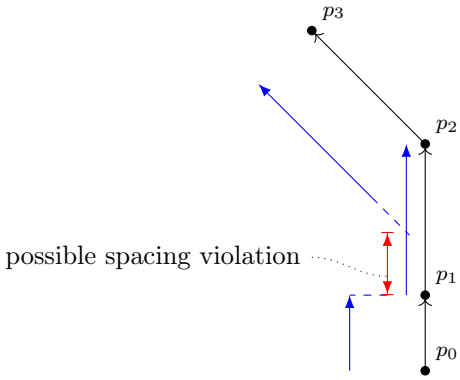
(a) Wires can reverse direction



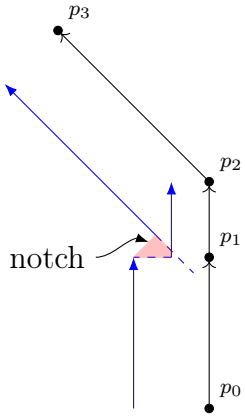
(b) Wires can reverse direction and cause loops



(c) Notches must account *multiple* small segments



(d) Distance to next intersection must be checked



(e) Angled corners can create notches

Figure 2.22: `PolyLine` complications

2.3.4 Layout Guides

BFG provides three layout "helper" classes to simplify common and repetitive layout task: an abstraction for rows of cells (`RowGuide`); an abstraction of banks of rows (`MemoryBank`); and an abstraction for grids where instances appear in every other slot (`CheckerboardGuide`). Each was born of necessity through the development of the generators in Chapter 4; they work by managing the absolute positions of their instances according to the size of each and a set of simple alignment constraints. Layout guides are the closest BFG comes to having an opinion on how layout should be assembled. They are offered both 1) as examples of how useful utilities can be created to divide and simplify layout into common tasks; and 2) as useful utilities in themselves. As in all software [23] [32], separating out such concerns from generator code and into special-purpose utilities makes complex generators tractable to write and easier to maintain.

2.3.4.1 Rows

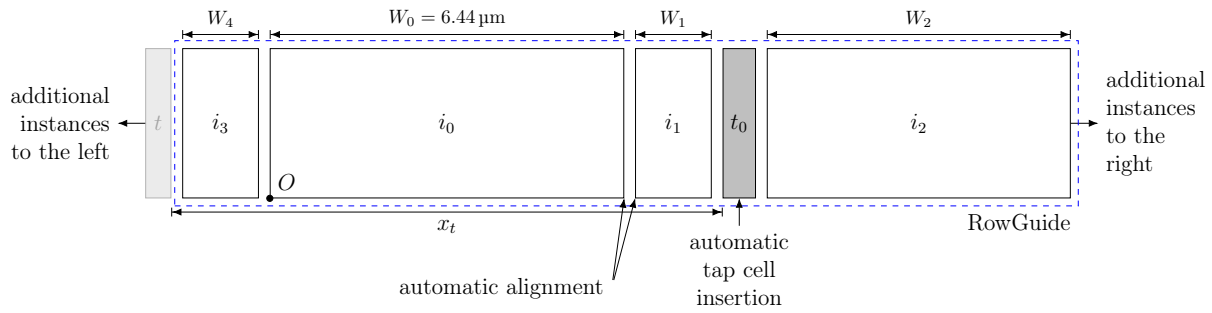


Figure 2.23: Row management with the `RowGuide` helper

In digital VLSI a design's entire floorplan (layout) is typically partitioned into contiguous equal-height and equal-width rows. The rows themselves are composed of equal-width slots, often called *sites*. It is common to rotate the cells of alternative rows by 180° so that the power and ground rails that are typically at top and bottom of the cells, as well as any N- or P-wells, line up. During placement, each standard cell is allocated to some number of such sites within a row.

Analog/mixed-signal designers can also benefit from a "row" abstraction, only with a relaxed set of constraints. Accordingly, BFG's `RowGuide` manages the horizontal[5] alignment of a sequence of cell instances. The class tracks the row's origin point, the alignment of cells of differing heights, and whether every instance in the row should be rotated. The entire row of cells can then be moved as a group.

[5]Vertical stacks could also be supported, in principle, by first rotating the instances in the row 270° and then rotating the horizontal stack 90°.

Users can add cells to the left (`RowGuide::InstantiateBack`) or right (`RowGuide::InstantiateFront`) and `RowGuide` will place them accordingly. To do this it determines their tiling boundary (or bounding box, if no tiling bounds are set), figures any difference in neighbouring cell heights according to specified alignment[6], and finally calculates the necessary position for the instance's origin. Consider Figure 2.23, where three instances $i_0, \ldots, i_2$ have been added to the right of a row, and a fourth i_3 has been added to the left, behind the origin. (Note that the horizontal spacing between the instances in the figure is illustrative only — there usually is none.) If the user instead inserts cells to the left with `RowGuide::InstantiateAndInsertFront`, all of the existing cells will be shifted to the right to make room for the new cell *at* the origin, O. Next, if the user specifies a template tap cell (Sec. 4.3.4) and sets the maximum distance between tap cells (x_t), the `RowGuide` will automatically insert tap cells as often as is required to keep their spacing below the requirement. If the user specifies that the entire row should be rotated, then every instance is rotated 180°.

2.3.4.2 Banks

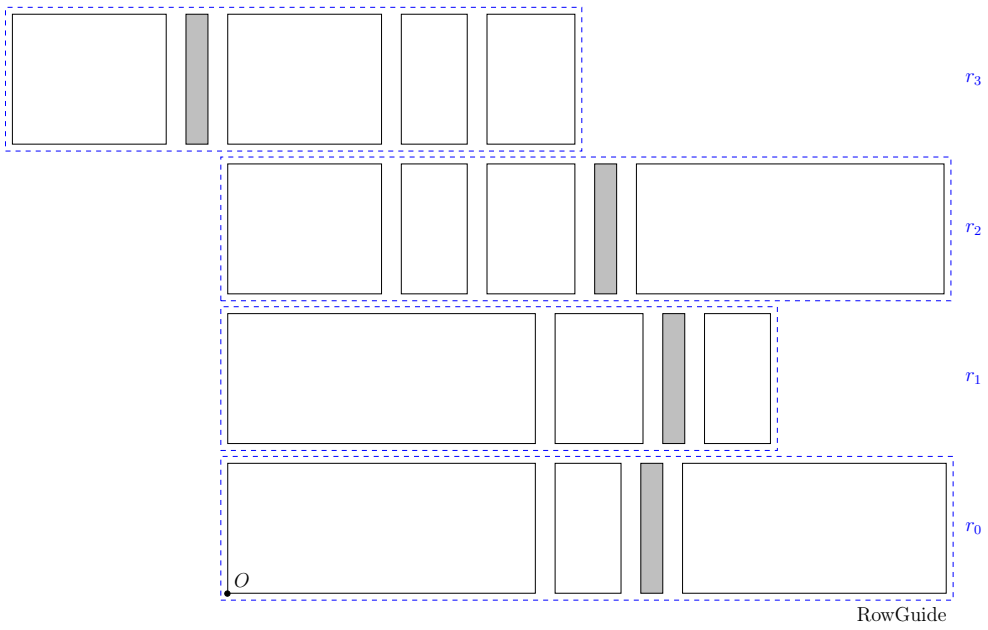


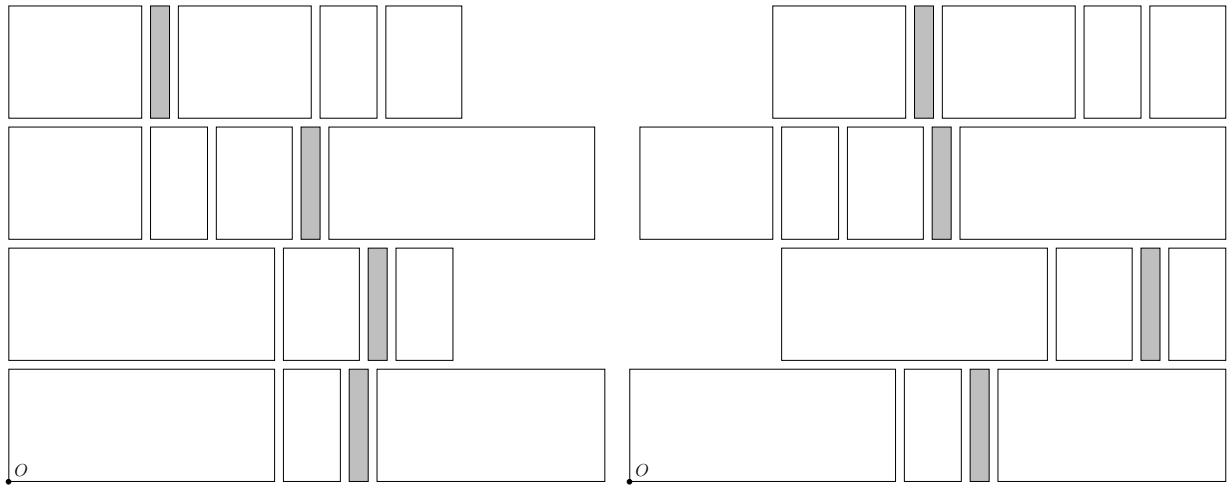
Figure 2.24: Multiple rows are managed together by the `MemoryBank` helper

The `MemoryBank`[7] class allows designers to conveniently manage a group of rows together. Consider Figure 2.24 (again, any horizontal spacing is illustrative, there is usually none).

[6]Cells do not need to be of the same height, but if they aren't an alignment must be specified. Currently only bottom-alignment, where cells' origins are set at the same vertical y position, is supported.

[7]BFG's banks were originally created to manage rows of memories, hence the name `MemoryBank`. It should be renamed in future work. Maybe.

The bank provides a convenient wrapper to its underlying collection of rows: new rows are automatically (lazily) created as they are referenced by the designer and configured according to a set of external options. Whether the additional rows are added at the top or at the bottom is such an option; whether alternating rows are rotated, whether the *first* row is rotated (so the parity of alternation is flipped), and what tap cells (if any) should be set on each row are also external options. Along with the tap cells, the bank also accepts a set of default connections to be made in the circuit-representation of the row as it is assembled. By default, rows manage the placement of each new instance relative to their own origin. The bank can be configured to enforce a horizontal alignment of rows, to the left (Fig. 2.25a) or to the right (Fig. 2.25b).



(a) Left-aligned rows (b) Right-aligned rows

Figure 2.25: Simple row alignment with `MemoryBank`

Listing 2.17 shows excerpts of the `tiles::LutB` generator where two banks are created (Sec. 4.4): one with left- and one with right-alignment. A number of memory cells are generated before an instance of each is assigned to a pre-determined row by index. If the row has not been referenced yet, it is created. We check for the bank alignment to determine whether the instance should be pushed to the left or right of the row, but this is just so that the memories line up where we want after the `MemoryBank` performs its alignment operation. The code is already somewhat verbose, but it is made much simpler for not having to consider any of the instance geometries.

Listing 2.17: An example of using the `MemoryBank` to simplify allocating memory instances to rows, based on excerpts from tiles/lut_b.cc

```
210 for (size_t p = 0; p < arrangements.size(); ++p) {
211   // BankArrangment tracks some details about the bank.
212   const BankArrangement &bank_arrangement = arrangements[p].get();
213   banks_.push_back(MemoryBank(layout.get(),
```

```
                                circuit.get(),
                                design_db_,
                                tap_cell,   // We generated this earlier.
                                true,  // Rotate alternate rows.
                                // Rotate the first row:
                                bank_arrangement.alternate_rotation,
                                bank_arrangement.horizontal_alignment));

  MemoryBank &bank = banks_.back();
  // Set default circuit connections for new tap cells.
  bank.default_tap_connections().insert({
      "VPWR", circuit->GetOrAddSignal("VPWR", 1)});
  bank.default_tap_connections().insert({
      "VFND", circuit->GetOrAddSignal("VGND", 1)});
  bank.default_tap_connections().insert({
      "VPB", circuit->GetOrAddSignal("VPWR", 1)});
  bank.default_tap_connections().insert({
      "VNB", circuit->GetOrAddSignal("VGND", 1)});

  // We now want to assign things to rows and have the memory bank create the
  // row if they don't exist.
  size_t num_memories = 0;
  for (size_t i = 0; i < bank_arrangement.memory_rows.size(); ++i) {
    size_t assigned_row = bank_arrangement.memory_rows[i];

    std::string template_name = /* A unique name for the cell. */
    std::string instance_name = absl::StrCat(template_name, "_i");
    atoms::Sky130Dfxtp::Parameters params = {/* Some cell parameters. */};
    // Generate the cell, and get a handle to it:
    atoms::Sky130Dfxtp generator(params, design_db_);
    Cell *cell = generator.GenerateIntoDatabase(
        PrefixCellName(template_name));

    geometry::Instance *installed = nullptr;
    if (bank_arrangement.horizontal_alignment == geometry::Compass::LEFT) {
      // Push an instance of 'cell' with the name 'instance_name' to the right
      // of row 'assigned_row'.
      installed = bank.InstantiateLeft(assigned_row, instance_name, cell);
    } else {
      installed = bank.InstantiateRight(assigned_row, instance_name, cell);
    }
    memories_.push_back(installed);
    ++num_memories;
  }
}
```

2.3.4.3 Checkerboards

Lastly, BFG provides a `CheckerboardGuide` utility class to help designers line up instances on a chequerboard grid[8], as shown in Figure 2.26. The main parameters to the `CheckerboardGuide` are:

[8]Laying out a complete grid is also possible with such a structure, but has not yet been necessary for our needs. `MemoryBank` can be used for an approximation of this already but, where regular arrays that require vertical *and* horizontal alignment are necessary, the `CheckerboardGuide` could trivially be extended to a "`GridGuide`" to support them.

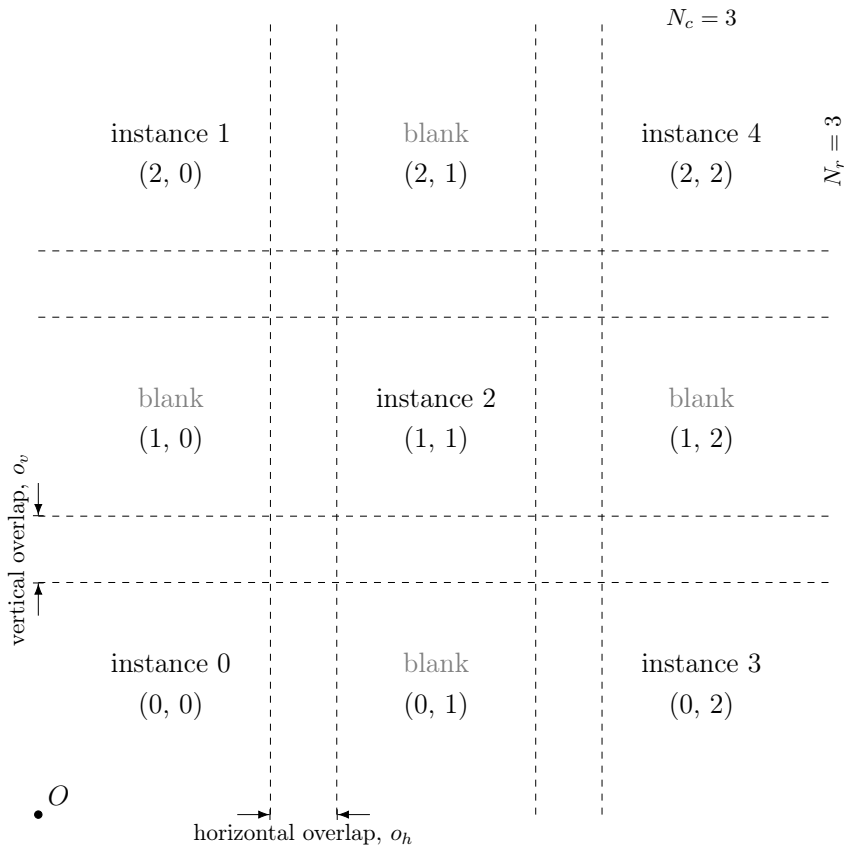


Figure 2.26: Checkerboard-style grid management with the `CheckerboardGuide`

1. the grid's origin O;
2. the size of the grid, in the number of columns N_c and the number of rows N_r;
3. a vertical overlap o_v between rows;
4. a horizontal overlap o_h between rows;
5. a circular buffer of N template instances $I = \{i_0, \ldots, i_{N-1}\}$ from which to allocate slots; and
6. a name prefix for each newly copied instance.

The `CheckerboardGuide` iterates over the grid in row-major order; every second grid position (even positions if starting at 0; odd if starting at 1) is considered to be a slot for an instance. Let k denote the slot number. We select instance i_k from the buffer of template instances I, wrapping at the end:

$$i_k \leftarrow I_{k \mod N}$$

Each template instance i_k is copied and given a unique name, generated by prefixing the original with the designer-nominated prefix value (if any) and then suffixing it with its position. The resulting copy is then translated to the appropriate origin given the grid's parameters and the instance's tiling bounds. This set up makes it simple to allocate the same set of instances to each row, to alternate between two types of cell (as we do in the `LutB` CLB generator, Sec. 4.4) or to use a single cell for the entire grid.

2.4 Process Portability

BFG affords process portability through two mechanisms. The first is to make it trivial to apply generators from one process to another. There exist classes of compatible processes among which layouts can have similar topologies, wherein the layout from one process will work with all of the others in that class after appropriate scaling is applied. This largely depends on how the process is developed, what lithographic steps are taken, and therefore on the categories of layout rules that apply. Not *every* valid layout topology for one process is expected to be valid for all of the others within these classes; rather, there should exist one layout style that works for all of them. We expect that this will result in compromised performance for some but, should that be important, someone can write a specialised generator for that process anyway.

We show that this principle works by re-using the `Sky130Mux` generator code to make `Gf180McuMux` in Sections 4.3.8 and 4.3.9. SkyWater 130 nm and Global Foundries 180 nm are two processes in such a class, and as such the layouts from SkyWater 130 nm will work for Global Foundries 180 nm if 1) an appropriate layer name mapping is provided and 2) the geometric (sizing and spacing) rules for Global Foundries 180 nm are used instead. Processes from newer FinFET nodes, by contrast, have discretised transistor widths (the number of fins). That might be adaptable, but newer nodes also tend to use multiple patterning, severely enforcing the regularity of, for example, polygon structures. Both Global Foundries 180 nm and SkyWater 130 nm support "snaking" horizontal and vertical polysilicon shapes, whereas modern processes might require unbroken, regularly spaced, aligned, and strictly vertical polysilicon to be placed. Dummy silicon might be required to satisfy patterning rules. Modern processes have significantly more complicated rule sets, but are still expressible in software.

The second mechanism is structural. Composing hierarchies of generators in the manner of BFG lends itself to abstraction. For example, some generators will implement the layout detail for a flip-flop in one process while other generators will implement it for another process. As one ascends the composition hierarchy, generators themselves should be designed around interchangeable process-specific parts that can be swapped in and out. If the flip-flops implement the same logical behaviour they should be logically interchangeable within some higher-level cell.

BFG explicitly codifies this kind of abstraction by delineating two kinds of generators: `Atoms`, which are specific to a single or a family of processes and their design topology;

and `Tiles`, which are intended to be process-agnostic higher-level compositions of `Atoms` and other `Tiles`. In practice, of course, this is not enforced by any mechanism within the framework. The designer is nonetheless encouraged to develop their solution this way. The CLB generator `tiles::LutB` (Sec. 4.4) demonstrates how higher-level generators might be used to implemented process-portability. It also demonstrates some of the pitfalls for doing so: often, for example, a upper-level layout will depend on the dimensions of and port positions of its child components, so success is not automatic. The BFG router(s) (Sec. 2.7), which try to abstract the problem of connecting such ports among child instances in a layout, try to address this problem.

2.5 Creating Circuits

In this section we will walk through the steps necessary to express a circuit in order to demonstrate how a designer would use BFG. We will use the example of SkyWater 130 nm's `sky130_fd_sc_hd__xor2_1` cell, which implements a 2-input exclusive-or gate. The schematic is shown in Figure 2.27.

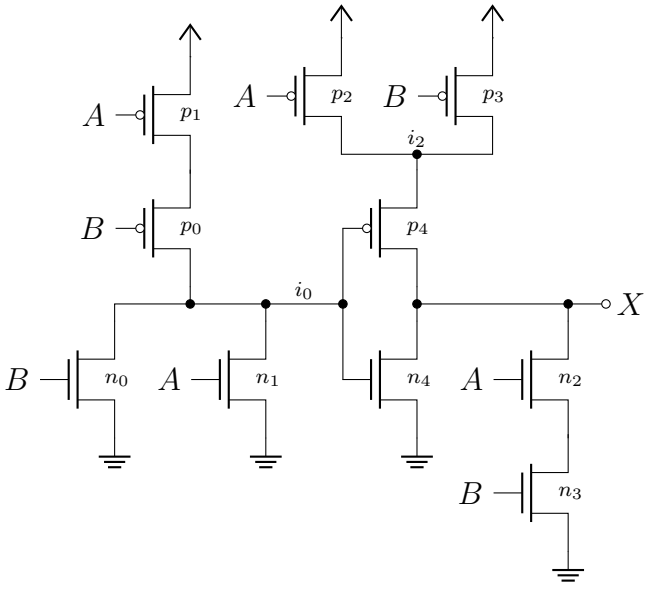


Figure 2.27: An XOR2 circuit

This is the `sky130_fd_sc_hd__xor2_1` circuit from Section 4.3.7.

The circuit has two signals for input logic A and B, an output X, and connections to power V_{DD} and ground V_{SS}. There are also two implicit connections, one to each of the NMOS and PMOS well substrates, which we respectively call V_{PB} and V_{NB}. (These follow from the convention of labelling the substrate terminal of a MOSFET as b.) A sensible —

but by no means mandatory — first step is to define these ports, which is done by first defining their associated signals and then indicating that they are ports:

```
bfg::Circuit *Sky130Xor2::GenerateCircuit() {
  std::unique_ptr<bfg::Circuit> circuit(new bfg::Circuit());
  circuit::Wire X = circuit->AddSignal("X");
  circuit::Wire A = circuit->AddSignal("A");
  circuit::Wire B = circuit->AddSignal("B");
  circuit::Wire VPWR = circuit->AddSignal(parameters_.power_net);
  circuit::Wire VGND = circuit->AddSignal(parameters_.ground_net);
  circuit::Wire VPB = circuit->AddSignal("VPB");
  circuit::Wire VNB = circuit->AddSignal("VNB");
  circuit->AddPort(X);
  circuit->AddPort(A);
  circuit->AddPort(B);
  circuit->AddPort(VPWR);
  circuit->AddPort(VGND);
  circuit->AddPort(VPB);
  circuit->AddPort(VNB);
```

Note that the names of the ports can themselves be parameters, because we are writing a generator. In this example, the power and ground nets are labelled according to values in some `parameters_` structure. Next, we look up the technology rules for the type of each transistor we want to use. We can change the transistor type to support any number of technologies, or variations of transistors within technologies. Here, we select regular 1.8 V NMOS transistors and High Threshold Voltage (HVT) 1.8 V PMOS transistors. In Section 4.7.1 we will see that transistors themselves can be abstracted to simplify this process.

```
  bfg::Circuit *nfet_01v8 =
      design_db_->FindCellOrDie("sky130", "sky130_fd_pr__nfet_01v8")->circuit();
  bfg::Circuit *pfet_01v8 =
      design_db_->FindCellOrDie(
          "sky130", "sky130_fd_pr__pfet_01v8_hvt")->circuit();
```

The circuit has five PMOS transistors $p_0, p_1, \ldots, p_4$ and five NMOS transistors $n_0, n_1, \ldots, n_4$, so we create an instance of each using a handle to the underlying transistor models we have just retrieved.

```
  // pfets.
  circuit::Instance *pfet_0 = circuit->AddInstance("pfet_0", pfet_01v8);
  circuit::Instance *pfet_1 = circuit->AddInstance("pfet_1", pfet_01v8);
  circuit::Instance *pfet_2 = circuit->AddInstance("pfet_2", pfet_01v8);
  circuit::Instance *pfet_3 = circuit->AddInstance("pfet_3", pfet_01v8);
  circuit::Instance *pfet_4 = circuit->AddInstance("pfet_4", pfet_01v8);

  // nfets.
  circuit::Instance *nfet_0 = circuit->AddInstance("nfet_0", nfet_01v8);
  circuit::Instance *nfet_1 = circuit->AddInstance("nfet_1", nfet_01v8);
```

```
circuit::Instance *nfet_2 = circuit->AddInstance("nfet_2", nfet_01v8);
circuit::Instance *nfet_3 = circuit->AddInstance("nfet_3", nfet_01v8);
circuit::Instance *nfet_4 = circuit->AddInstance("nfet_4", nfet_01v8);
```

We can now encode the topology. In AMS designs, signals are typically single-width wires that have multiple connections, so we can use the `circuit::Wire` class to maintain simple named references to them. In digital designs, by contrast, signals are just as often wide buses, can be sliced and concatenated, and must be managed accordingly. We must create a signal for each shared wire (node), such as i_0 and i_2 in the schematic, and also for the direct connections with only two terminals. We note this as a weakness of the interface; it would simple and preferable to support an interface that automatically creates intermediate signals when two or more instance ports are connected together. (This is the approach of Hdl21 [30] and others.)

Connections are issued by naming the ports of each transistor instance and the shared signals. They can be specified one port at a time, or in bulk:

```
circuit::Wire i_0 = circuit->AddSignal("i_0");
nfet_0->Connect({{"d", i_0}, {"g", B}, {"s", VGND}, {"b", VNB}});
nfet_1->Connect({{"d", i_0}, {"g", A}, {"s", VGND}, {"b", VNB}});

circuit::Wire i_1 = circuit->AddSignal("i_1");
pfet_0->Connect({{"d", i_1}, {"g", B}, {"s", i_0}, {"b", VPB}});
pfet_1->Connect({{"d", VPWR}, {"g", A}, {"s", i_1}, {"b", VPB}});

circuit::Wire i_2 = circuit->AddSignal("i_2");
pfet_2->Connect({{"d", VPWR}, {"g", A}, {"s", i_2}, {"b", VPB}});
pfet_3->Connect({{"d", VPWR}, {"g", B}, {"s", i_2}, {"b", VPB}});

nfet_4->Connect({{"d", X}, {"g", i_0}, {"s", VGND}, {"b", VNB}});
pfet_4->Connect({{"d", i_2}, {"g", i_0}, {"s", X}, {"b", VPB}});

circuit::Wire i_3 = circuit->AddSignal("i_3");
nfet_2->Connect({{"d", X}, {"g", A}, {"s", i_3}, {"b", VNB}});
nfet_3->Connect({{"d", i_3}, {"g", B}, {"s", VGND}, {"b", VNB}});
```

The last step is to set the parameters of the transistors. We must set the widths and lengths of our transistors to match the parameters given to the layout generator, so that the circuit and layout models correspond. The designer is free to express this as they wish. In this example, we collect all of the parameters from the cell parameter structure into a local array:

```
struct FetParameters {
  circuit::Instance *instance;
  uint64_t width_nm;
  uint64_t length_nm;
};
std::array<FetParameters, 10> fet_parameters = {
```

```
    FetParameters {
      nfet_0, parameters_.nfet_0_width_nm, parameters_.nfet_0_length_nm
    },
    FetParameters {
      nfet_1, parameters_.nfet_1_width_nm, parameters_.nfet_1_length_nm
    },
    /* ... and so on, for each transistor. */
    FetParameters {
      pfet_4, parameters_.pfet_4_width_nm, parameters_.pfet_4_length_nm
    }
  };
```

and use that to assign all corresponding parameters in a single loop:

```
  for (size_t i = 0; i < fet_parameters.size(); ++i) {
    circuit::Instance *fet = fet_parameters[i].instance;
    fet->SetParameter(
        parameters_.fet_model_width_parameter,
        Parameter::FromInteger(
            parameters_.fet_model_width_parameter,
            static_cast<int64_t>(fet_parameters[i].width_nm),
            Parameter::SIUnitPrefix::NANO));
    fet->SetParameter(
        parameters_.fet_model_length_parameter,
        Parameter::FromInteger(
            parameters_.fet_model_length_parameter,
            static_cast<int64_t>(fet_parameters[i].length_nm),
            Parameter::SIUnitPrefix::NANO));
  }
```

In the case of foundry devices such as these, the *names* of the parameters that correspond to the *width* and *length* of the device, or perhaps the number of fins it has (in FinFET devices), are defined by the foundry PDK. Since these are common to any generator targeting the technology, we can define a common parameter superclass that makes these available to all of them:

```
struct Sky130Parameters {
  static constexpr uint64_t kStandardCellUnitWidthNm = 460;
  std::string fet_model_length_parameter = "l";
  std::string fet_model_width_parameter = "w";
  bool draw_vpwr_vias = true;
  bool draw_vgnd_vias = true;
  std::string power_net = "VPWR";
  std::string ground_net = "VGND";
};
```

These could even be read directly from the foundry PDK or as part of the technology model; whatever designers consider most convenient. A still better abstraction is to generalise the transistor cell itself (Sec. 4.7.1), but this is a simpler demonstration of the machinery. Once

we have found the appropriate parameter, we assign its value using the `Parameter` struct (see `parameter.h`), which explicitly pairs integral values with SI units (nano, pico, femto, and so on). This way both a) we capture the designer's intent and b) there is never any loss of precision.

```
  return circuit.release();
}  // End of Sky130Xor2::GenerateCircuit()
```

With that, the circuit is complete, and we can move onto layout.

2.6 Creating Layout

In this section we will briefly demonstrate how BFG can be used to create layout. Layout code grows complex quickly, as any of the generators in Chapter 4 will attest. The reader is encouraged to read the code for BFG's generators in the `atoms/` and `tiles/` subdirectories of the source repository for a more thorough understanding of what is involved. Here, for illustrative purposes, we will walk through the creation of the simple PMOS transistor show in Figure 2.28.

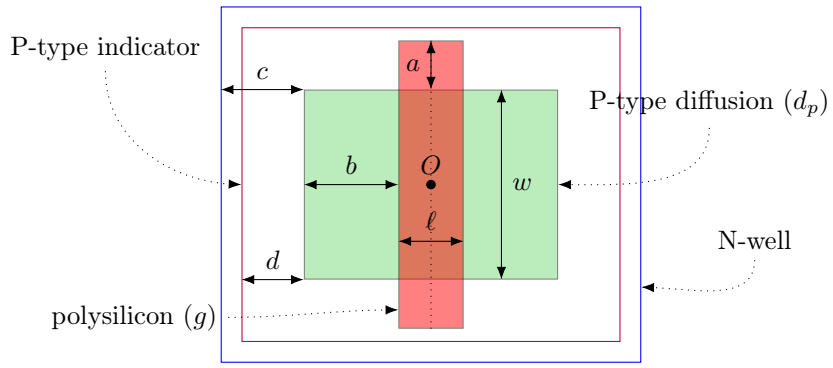


Figure 2.28: Plan for a transistor layout

We start with handle to the `PhysicalPropertiesDatabase` and a `Layout` object:

```
bfg::Layout *LayoutDemo::GenerateLayout() {
  const PhysicalPropertiesDatabase &db = design_db_->physical_db();
  std::unique_ptr<bfg::Layout> layout(new bfg::Layout(db));
```

The width w and length ℓ of our transistor will be set by the parameter struct that our generator class takes as an argument. In our case they are set to $w = 500\,\text{nm}$ and $\ell = 170\,\text{nm}$, which we will assume are reasonable (legal) values. Since these are specified in physical units, we must convert them to the internal units of our database:

```
int64_t width = db.ToInternalUnits(parameters_.width_nm);
int64_t length = db.ToInternalUnits(parameters_.length_nm);
```

In addition to the width and length, we need to be factor in two technology constraints: the minimum distance the polysilicon must extend past the diffusion region (span a in Fig. 2.28), and the minimum distance the diffusion must extend past the polysilicon in the perpendicular direction (span b in Fig. 2.28). (Note that this is the same example of a non-commutative design rule shown in Section 2.2.1 Figure 2.3.) We can get both at the same time by looking up the *inter-layer* rules between `pdiff.drawing` and and `poly.drawing`. Incidentally, these layer names come from the foundry PDK; the designer must consult the PDK's documentation in conjunction with BFG's input technology files to determine what names and aliases are available. The user can also customise these further if they wish.

```
const auto &poly_diff_rules = db.Rules("poly.drawing", "pdiff.drawing");
int64_t poly_overhang = poly_diff_rules.min_enclosure;
int64_t diff_wing = poly_diff_rules.min_extension;
```

The overall width w_d and height h_d of the diffusion box are, per Figure 2.28, given by

$$w_d = 2b + \ell$$
$$h_d = w$$

Creating the shape is a matter of calculating the necessary dimensions and instantiating the appropriate geometry object from the library. The default `geometry::Rectangle` constructor takes the lower-left and upper-right `geometry::Points`. If we imagine the centre of the transistor at the origin, these points become $\pm(w_d/2, w_d/2)$. We set the `layout` object to the diffusion layer and add the rectangle:

```
layout->SetActiveLayerByName("pdiff.drawing");
Rectangle *diff = layout->AddRectangle(Rectangle(
    Point(-length / 2 - diff_wing, -width / 2),
    Point(length / 2 + diff_wing, width / 2)));
```

In SkyWater 130 nm, P-type diffusion is defined by the union of two layers: generic `diff.drawing` and `psdm.drawing`. We must add a `psdm` box to make sure the diffusion is interpreted correctly. A separate technology rule governs the minimum enclosure of diffusion and `psdm.drawing` (d in Fig. 2.28). After looking this up in the `db` we could calculate the bounds of a new rectangle, or use simply use the `Rectangle::WithPadding` method instead:

```
int64_t psdm_enclosure =
    db.Rules("pdiff.drawing", "psdm.drawing").min_enclosure;
layout->AddRectangle(diff->WithPadding(psdm_enclosure));
```

Similarly, P-type diffusion is typically implanted into an N-doped region of an otherwise P-type substrate. The N-doped region is called the *N-well*. We look up the enclosure rules

to figure out how big the N-well needs to be and create another box, except this time we do it the complicated way to demonstrate the benefits of convenient APIs like `Rectangle::WithPadding`:

```
  int64_t nwell_enclosure =
      db.Rules("pdiff.drawing", "nwell.drawing").min_enclosure;
  Point diff_ll = diff->lower_left();
  Point diff_ur = diff->upper_right();
  layout->SetActiveLayerByName("nwell.drawing");
  layout->AddRectangle(Rectangle(
      Point(diff_ll.x() - nwell_enclosure, diff_ll.y() - nwell_enclosure),
      Point(diff_ur.x() + nwell_enclosure, diff_ur.y() + nwell_enclosure)));
```

The polysilicon gate is another rectangle with width w_p and height h_p dictated by the transistor width and length as well as the minimum extension of polysilicon over diffusion, a, that we found earlier:

$$w_p = \ell$$
$$h_p = 2a + w$$

By this point we know how to create rectangles. Instead we will use a `PolyLine`, following the dotted centreline of the box shown in Fig. 2.28, and setting its width. Because we have centred the transistor on the origin, the two points defining the line are $\pm(0, {}^{h_p}/_2)$. We can also use `ScopedLayer` to manage the layer on the `layout` for us, tidying up the code somewhat.

```
  {
    ScopedLayer sl(layout, "poly.drawing");
    PolyLine gate({Point(0, -width / 2 - poly_overhang),
                   Point(0, width / 2 + poly_overhang)});
    gate.SetWidth(length);
    layout->AddPolyLine(gate);
  }
```

The last thing to do is to save a few named points of interest and return the finished layout. As `Sky130SimpleTransistor` does, we could at this point calculate where on the diffusion region it would be possible to connect to with a via to the metal layer above. This is an easy mechanism for calling code to use to re-discover those points without having to re-calculate all of their geometries.

```
  layout->SavePoint("diff_lower_left", diff->lower_left());
  layout->SavePoint("diff_upper_right", diff->upper_right());
  layout->SavePoint("gate_bottom_centre", Point(0, -width / 2 - poly_overhang));
  layout->SavePoint("gate_top_centre", Point(0, width / 2 + poly_overhang));

  return layout.release();
}
```

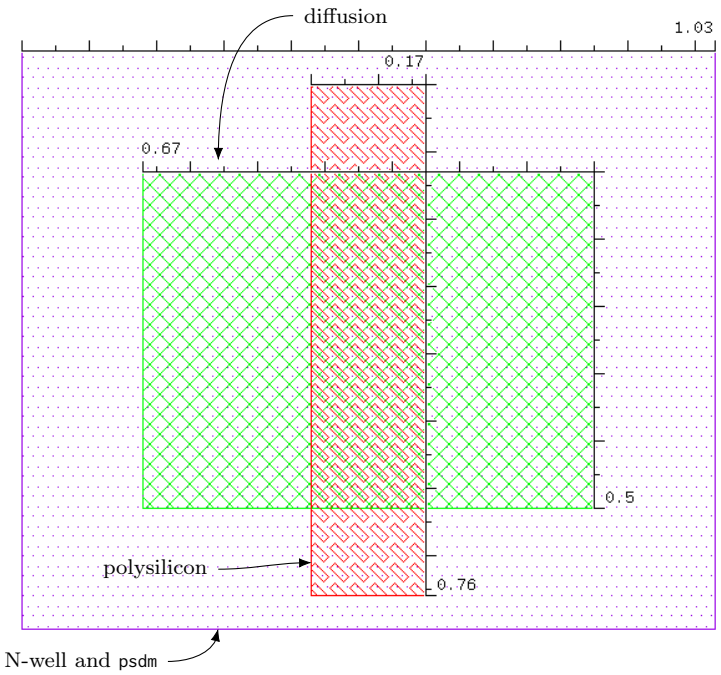


Figure 2.29: Exported GDS of the demonstration transistor
The units in the screenshot are micrometres, µm.

On export and conversion to GDS-format, a popular viewing tool like kLayout will check our work, as shown in Figure 2.29. Note that the N-well and `psdm` layer are the same size and shape.

2.7 Automatic Routing

BFG provides a facility to automatically lay wiring when crafting IC layout. As one ascends the cell hierarchy, the number of wires that must be simultaneously negotiated grows intractably large. Likewise, as the number of parameters to which a cell must adapt grows, procedurally specifying paths that can respond to all of them becomes intractably complex. There may still be critical paths that are worth careful procedural optimisation (BFG is unopinionated and the router is an optional feature) while still others merely need a viable option.

There are many established approaches to automatic routing, especially in the context of IC layout. We use ideas from the large-scale global routers of standard digital flows (Sec. 1.3.2) and the small-scale automatic layout of leaf cells (Sec. 1.7.4). Our router is designed to a minimum-viable tool for the designer of AMS circuits and layout. It should allow the architect to place a number of cells and route them relatively quickly (taking on the order of minutes, not hours).

2.7.1 Implementation

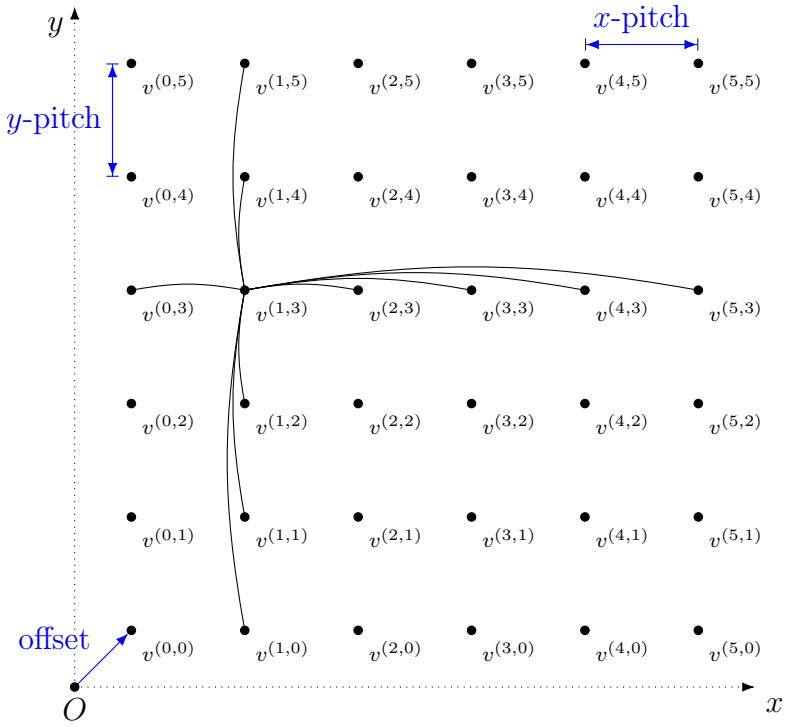


Figure 2.30: The `RoutingGrid` model

BFG's router is implemented by `RoutingGrid` and a host of related classes. Like most routers, `RoutingGrid` works on an undirected graph-based representation of legal wiring positions within the layout space. Wire segments can travel vertically or horizontally, never diagonally. Horizontal travel is generally restricted to one layer of layout, while vertical travel is restricted to another. Wire paths connect these layers with vias when changing direction. Letting $G = (V, E)$ denote the routing graph in this model, the vertices $v \in V$ of the graph (class `RoutingVertex`) represent the points at which vias could be placed and, therefore, where the direction can change. The edges ($e \in E$) of the graph (class `RoutingEdge`) represent possible wire segments. Wires, which are solutions to routing tasks and which are represented by the `RoutingPath` objects, are represented by an ordered list of vertices and edges $(v_0, e_0, v_1, e_1, v_2, \ldots, v_{N-1}, e_{N-1}, v_N)$.

Non-linear Cost Graph Model

The underlying routing model is illustrated in Figure 2.30. Within the routing area boundary, vertices are placed at regular intervals in the horizontal and vertical directions according to the x- and y-pitch specified by the caller. The entire grid can be shifted from the origin O

by specifying an *offset*. We call each row and each column a *track*. The `RoutingTrack` class maintains references[9] to the vertices on that track and is responsible for the edges between them. Edges are created between *every pair* of vertices within a track: Figure 2.30 shows all of the horizontal and vertical edges in which vertex $v^{(1,3)}$ participates.

In a popular alternative formulation of the routing graph $G' = (V', E')$, edges $e' \in E'$ only appear between vertices $v' \in V'$ that are immediate neighbours. Vertices $v' \in V'$ occupy a single layer. Whereas in the first formulation G a vertex in a path *implied* the existence of a via when straddled by edges on different layers, here the via is represented by an explicit edge between a vertex above and one below. Though the alternative model has its advantages, it restricts wire models (which dictate the cost associated with each edge e') to be linear. We know that the electrical delay (τ_{DN}) of a signal propagating through a wire is proportional to the square of its length (see Chapter 4 in [78]):

$$\tau_{DN} = \frac{rcL^2}{2}$$

under a distributed-rc model where r is the resistance per unit length, c the capacitance per unit length, and L the total length of the wire. Because the edges span the full length between vias, BFG's routing model can easily represent cost functions that are non-linear in wire length without having to modify existing path-finding algorithms (if that is even feasible). We refer to the alternative formulation as the *linear* model.

The main advantage of the linear model is that it is fast. For an $N \times M$ grid of vertices the linear model generates

$$M(N-1) + N(M-1) + NM \approx \mathcal{O}(NM)$$

edges, whereas BFG's model generates

$$M(N(N-1)) + N(M(M-1) \approx \mathcal{O}(MN^2 + NM^2)$$

edges. For an $N \times N$ grid these become $\mathcal{O}(N^2)$ and $\mathcal{O}(N^3)$ respectively. The linear model scales memory usage better in the number of vertices *and* reduces the computational complexity at each vertex, since there is a factor of N fewer edges to follow. Accordingly, BFG's router *also* supports a linear model, but this is largely untested and left for future work.

At a high level, finding valid paths is just a matter of tracking the so-called *validity* of all vertices and edges and then using a shortest-path algorithm to connect a given start and a given end point. Once a valid path is found it can be *installed*, a bookkeeping task whereby its constituent vertices and edges are invalidated for future use. The validity of a vertex or edge, however, depends as much on *who* is using it and for *what* as it does on whether it's being used at all. The majority of the complexity in the router centres on determining vertex validity.

[9] In this formulation, vertices are referenced by a horizontal and a vertical track but are owned by the parent `RoutingGrid` object.

Since wires (and thus paths) represent connectivity to a signal (or node) in the electrical circuit being drawn, paths are equipped with a *net* label that indicates their corresponding electrical node by name. All other shapes in the layout model already have net labels (Sec. 2.2.3). In addition to point-to-point routes, The BFG router supports finding routes from a starting point to any existing shape on a given net. We must expand our notion of validity accordingly: a vertex should be valid for connection if it is in use by a net matching the target one and on the right layer.

Obstructions

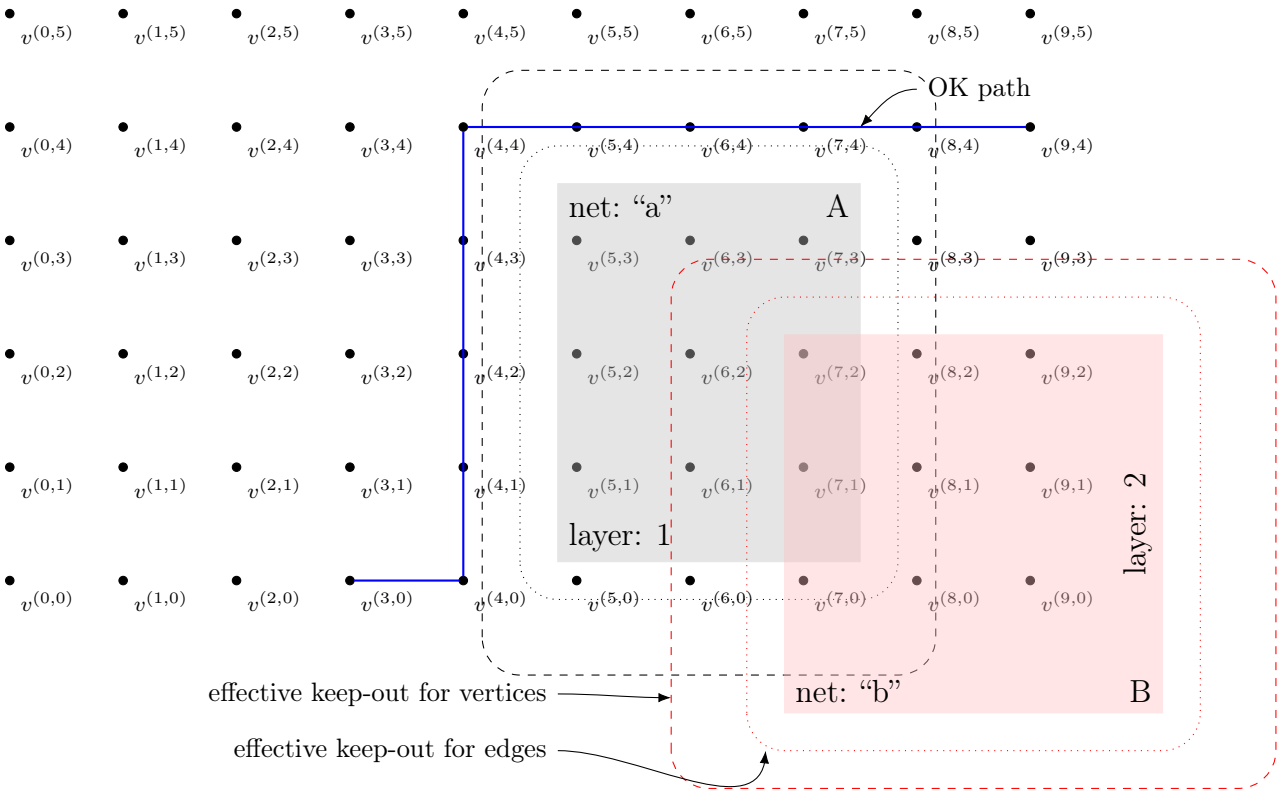


Figure 2.31: Obstacle avoidance

Routing is usually performed after a handful of other shapes have been laid, so the router must be able work around them. The `RoutingGrid` allows callers to declare existing shapes as *obstructions* using `ShapeCollections` or `Groups`. Consider Figure 2.31, where there are two obstructions A and B on the lower and upper routing layers, respectively. The router uses the minimum spacing rules for an obstruction's layer to establish a boundary around it (this is the "keep-out" area, see dashed boxes in the figure). These are `RoundedRectangles` because

the minimum separation distance defines an arc from the obstruction's corners. Any vertices that overlap the keep-out area are marked as *in-use* by the net of the respective obstruction. Vertices accumulate obstructions: $v^{(7,2)}$, for example, would be marked as in-use by both nets "a" and "b".

Note that the (rectangular) footprint of a via placed on a vertex depends on the direction of the wire entering or exiting it (recall Figure 2.4), and that the direction of the wire depends on the layer as well, therefore the padding of the keep-out region is not the same on all sides. In reality, it is the via footprints that are padded to a keep-out `RoundedRectangle` and that are then used to test intersection with obstruction shape, rather than the other way around. This because generating the keep-out shape for a rectangle (a rounded rectangle) is trivial, but generating the keep-out shape for a polygon (which the obstructions might be) is not. Maybe the diagram should just show this to begin with? Does it really help understanding to start the other way around?

What will eventually be labelled as a single net can, for the duration of layout compilation, have many different names. This is especially true as layouts are combined from different cells, where internal structures have their own labels. It would be burdensome to force the caller to relabel all relevant shapes with the appropriate nets; it is much easier to provide an equivalent class, `EquivalentNets`, that will manage them all. As nets are associated become equivalent to one another (through connection), they are added to an `EquivalentNets` object that can then be used to test for the equivalence of any other string (or even other `EquivalentNets` object). The class additionally stores a "primary" net name that will be used to generate the final label at export. This also explains why nets are labelled by name and not by reference to some `circuit::Signal`. BFG gives the designer flexibility to compose circuits and layouts independently, as long as they are self-consistent and *eventually* concordant. Using references to `circuit::Signal` objects would require pervasive consistency and possibly infuriate the designer.

Obstructions apply to edges in a different manner than they do to vertices. They are constrained by interactions on different sets of layers: vertices must accommodate a via and the encapsulating metal pours on upper and lower layers, while edges are just minimum-width wires. In the figure, vertex $v^{(5,4)}$ could only be used to connect to net "a", whereas an edge on the same horizontal track, such as $(v^{(3,4)}, v^{(9,4)})$, might be able to fit just passing by. A second margin is calculated using the configured wire thickness (per layer) and applied to the obstructions (the dotted lines in the figure). This is then used to mark edges as in-use by the appropriate nets (and on the appropriate layers). As a result, the path $(v^{(3,0)}, v^{(5,0)}, v^{(4,4)}, v^{(9,4)}$ would fit around the obstructions.

Obstructions can be permanent or temporary. Permanent obstructions arise from existing shapes that will persist throughout the lifetime of the `RoutingGrid`, and are fed directly to the router using `RoutingGrid::ApplyBlockages`. All of the obstruction information is then recorded directly onto the affected vertex and edge objects. Temporary obstructions are defined *per path search* and removed after the search concludes. They are typically used to avoid blocking some access ports while routing others. Without this mechanism, there is no way to ensure that a set of routes does not immediately preclude a future route by blocking

its start or end points.

Parallelism and Spatial Lookups

BFG provides two bookkeeping methods for tracking obstructions. The first is to simply mutate the objects directly, as we have already seen. This presents a barrier to parallelising (multi-threading) the path search process. It would not be possible to correctly run simultaneous searches with different sets of temporary blockages. To resolve this, BFG provides a blockage cache mechanism through `RoutingBlockageCache` that processes obstructions in the same manner but then stores the results as *references* to the which vertices and edges are affected. Each of a number of threads can then undergo a path search with its own cache of temporary blockages under a shared read-only lock.

Obstruction processing is computationally expensive. To ameliorate this, `RoutingGrid` uses the simple bounding box of an obstruction to quickly limit which vertices should be checked. The `RoutingGridGeometry` utility class is responsible for mapping designer-specified routing bounds, pitch, and offset into the coordinates and ordinal positions of each `RoutingTrack` and `RoutingVertex`. This enables it to quickly map a given shape to the set of vertices which surround it. (See the `RoutingGridGeometry::EnvelopingVertices` method.) Off-grid vertices, which are not tracked by `RoutingGridGeometry` but which are discussed below, are stored in a k-dimensional tree ($k = 2$) — a rare but welcome external library — for rapid lookup given some spatial bounds.

The BFG router is serial in nature; routes can be found in parallel but must be installed in some order, because the installation of one route might invalidate another. Solver-based methods such as those used in [43] (ILP, MIP, SAT) solve for many routes simultaneously, but at great runtime cost. In keeping with its design philosophy, BFG gives designers a minimum set of relatively basic tools that can be composed with great control and to great effect. Serial routing, as in BFG, is the basis of the tried-and-true Rip-up and Reroute method of more sophisticated general-purpose routers, which could be built on top. (These routers are built for orders of magnitude many more routes, but could even perhaps be finagled into BFG as they are.) This is also true for other methods, such as simulated annealing and the ILP, MIP, SAT (and even QP) solvers. Alas, such improvements are left to future work (Ch. 5). As a simple first step, `RoutingGrid` offers the `AddBestRouteBetween` and `AddBestRouteToNet` methods which take multiple start points and multiple end points as inputs. The router searches for the shortest path between all start and end pairs, and returns the lowest cost one.

Off-grid points

It is common for VLSI routers to require that source and destination points fall onto at least one track of the routing grid, and for good reason. BFG does not. BFG allows routes from any point in the space to any other. We did this to reduce the limitations on layouts that can be routed by BFG, in the hope of making the designer's life easier. As we shall see, this

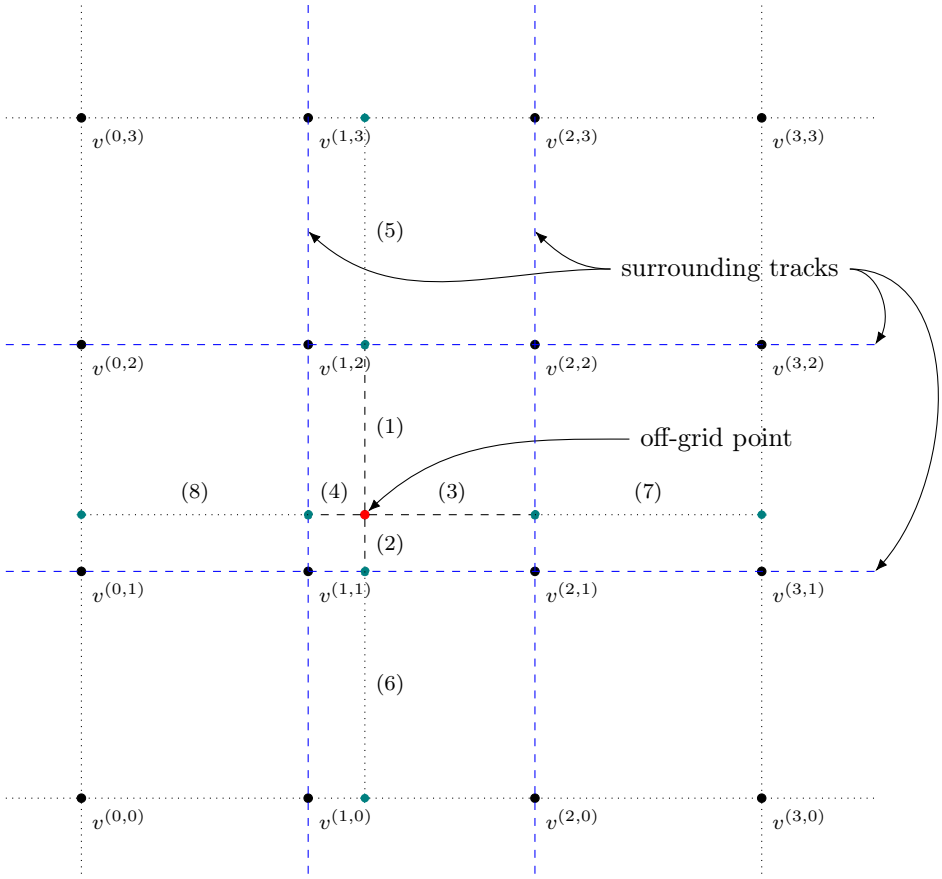


Figure 2.32: Connecting an off-grid vertex to nearby tracks

feature mostly creates complications for the router itself. The router must connect off-grid points to the grid before it can proceed. It tries two methods. The first, shown in Figure 2.32, is to check if any of the surrounding tracks can accommodate connections without design-rule violations. Up to two layers of enveloping tracks are checked and nearer tracks are preferred; in the figure, the off-grid edges (5) - (8) are only attempted if edges (1) to (4) fail. If a via and appropriate edges can be accommodated, the router creates an off-grid `RoutingVertex` and a `RoutingEdge`. These must be managed separately to those within the regular grid, which is itself a bureaucratic headache.

The second method is to find the nearest connectable vertex, as shown in Figure 2.33. This is a last resort, and exacerbates the two central complications that stem from allowing off-grid vertices: 1) connections to the grid, and subsequently the entire wire path, may not be optimal because they are made on a heuristic best-effort basis; and 2) the additional vertex and edges may cause more conflicts than they solve, making the overall problem of determining a legal set of routes harder. In Figure 2.33, the fallback method finds a connectable vertex reasonably far away (imagine that nearer vertices are blocked from hosting

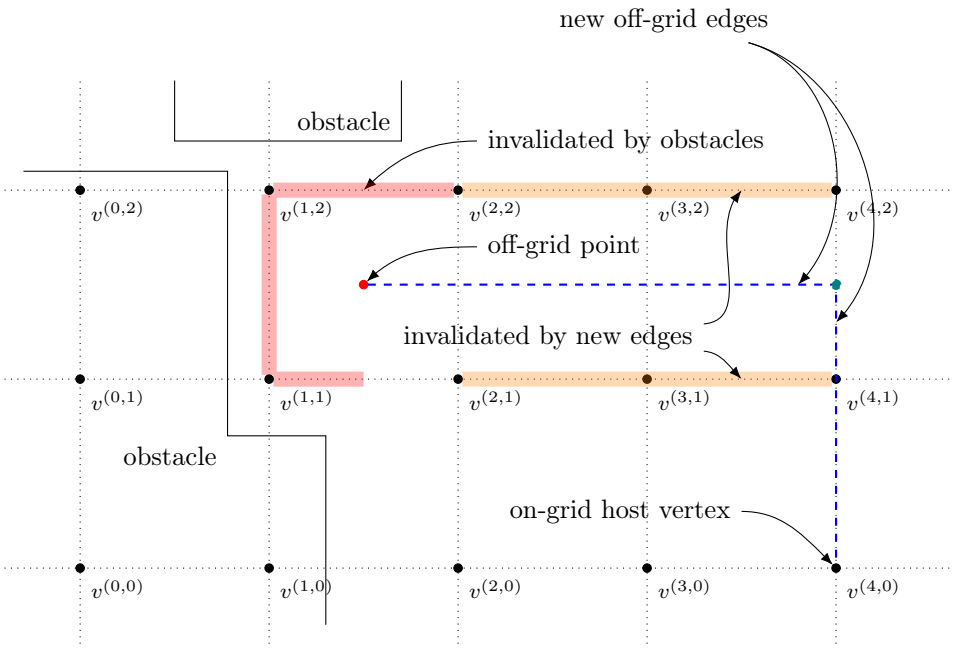


Figure 2.33: Connecting an off-grid vertex to the nearest possible point

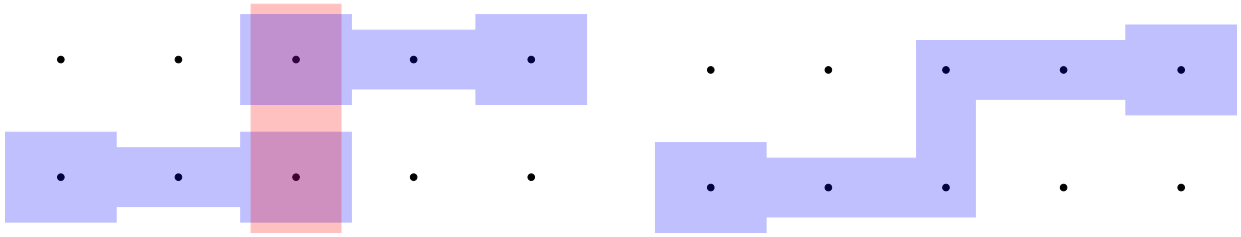
vias), but requires travel to its track in order to land on the grid. This off-grid edge then invalidates *two* neighbouring tracks' worth of nodes. Off-grid edges violate the invariants of a routing grid (which is otherwise sized to minimise collisions), and result in constant additional checks. There is a trade-off between optimising the connection to the grid, and minimising the disturbance to the grid as a result of it.

Shortest-Path and Legalisation

Most of the complexity in the router has been in the express service of setting-up a well-defined problem for the routing algorithm(s), which are richly described in literature. Once the source and destination points are connected to the routing grid, Dijkstra's shortest-path algorithm is used to find lowest-cost path between them. In the alternate mode, a slight tweak to the algorithm finds the lowest-cost path to any vertices that are on a specific net. With still another slight tweak, Dijkstra's algorithm could (and should) be turned into the A* algorithm, which uses a secondary heuristic to order the search for a quicker solution. Given that the routing grid is regular, a simple L2-norm (Euclidean distance) heuristic would work well. Again, these are well studied problems and there are many possible improvements over BFG's minimum viable solution.

A notable and lingering concern for the router is *determinism*. Designers typically wish for solutions to be stable; if the inputs provided to the router do not change then the outputs should not change either. Heavy-weight VLSI routers tend to accept a *seed* value

to set up whatever random processes are necessary, so that users can both exploit and avoid stochasticity. BFG should eventually do the same, but does not today. The central source of randomness in BFG is in memory layout by the operating system: it is fast and efficient to store object pointers in data structures like a `std::set`[10], but by default these pointer values are used as sort keys. In some places (e.g. `PortSet`) a comparator is provided to avoid this, but this has not been implemented pervasively. We leave this to future work.



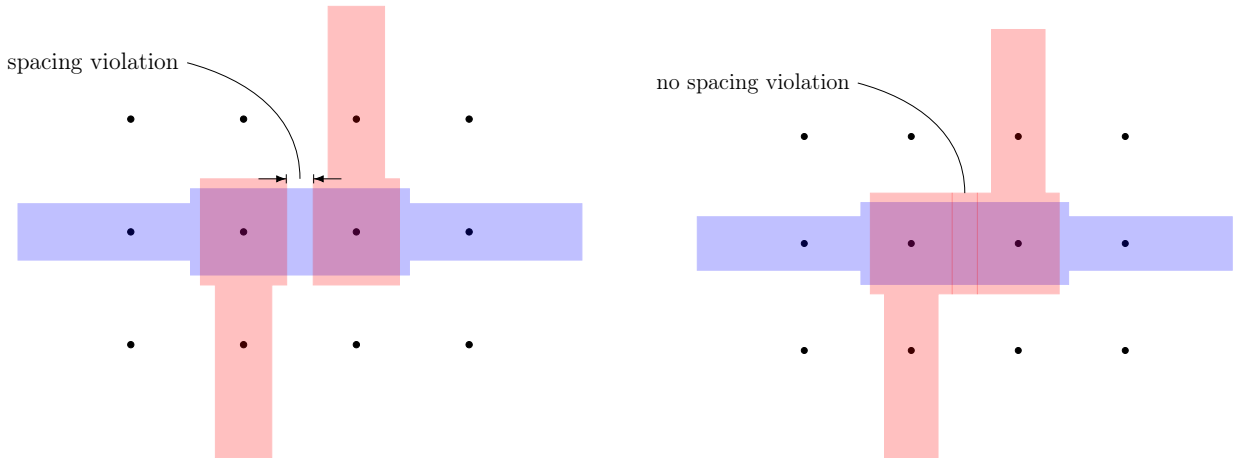
(a) Router makes correct but illegal track switch (b) Track switch corrected with jog

Figure 2.34: Post-route optimisation legalises tracks

After a suitable path is found, but before it is installed, a series of peep-hole optimisations are performed to try to correct oddities of the grid-based routing model into more sensible layout. It is perfectly correct according to the rules of the model, for example, to generate a path that jumps between two adjacent tracks by switching to the perpendicular layer (Fig. 2.34a). However, not only does this wire incur the unnecessary resistance of traversing two needless vias, it probably also violates spacing rules (as it does SkyWater 130 nm). `RoutingPath::Flatten` is invoked to detect such occurrences — while the path defines the wire in simple abstract terms — and replace them with a simple jog between the wires on the first layer. The result is both legal according to process design rules and a faster electrical path. Most touch-ups are dispatched by the `RoutingPath::Legalise` method. A more general version of this `RoutingPath::Flatten`, `RoutingPath::Abbreviate`, tries to make similar simplifications to a less obvious wire. The arduous details can be found in `geometry/routing_path.cc`.

Other mishaps are patched by the `RoutingGrid` itself, *after* all routes are installed (installation is discussed next). When multiple routes targeting a *net* find their way to adjacent vertices on an existing path, for example, they can end up cause a spacing rule violation between their via encapsulations (Fig. 2.35a). The receiving `PolyLine` path will automatically join two encapsulations together on its layer if they are too close, but no single object is aware of the encapsulations on the two arriving paths. In another pass over the solution the violation is resolved with a simple patch between the encapsulations (Fig. 2.35b). The approach to fixing this resembles the work of a basic DRC checker, because the canonical source of truth is now the set of all shapes and their geometric relationships. Until now

[10] `std::set` is part of the C++ standard library.



(a) Two wires meeting at an existing path

(b) A simple patch is needed

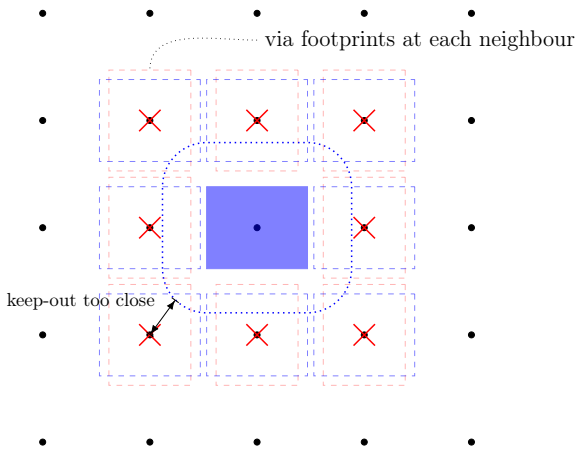
Figure 2.35: Pre-export optimisation patches oddities

such violations could only have been avoided *indirectly*[11]. Other optimisations are desirable, chief among which is a general simplifying pass that can reduce the number of layer changes through vias, perhaps by swapping edges between layers after they are all set. It should already be easy to find edges in a path, which intersect no other edges, and whose layer can be raised or lowered to that of antecedent or subsequent edges. These will help bridge the gap between the router's serial approach and simultaneously-optimising (ILP, QP, MIP, SAT) solver approaches (sec. 1.7.4).

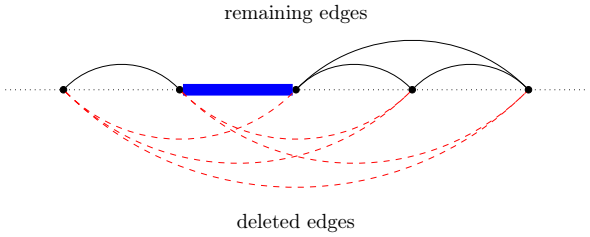
Installation and Export

Installation is the process of accepting a routing solution and committing it to the state of the model. The component vertices and edges of the path must then be marked as used. In practice, this is process is similar to accounting for obstructions. On- and off-grid vertices which host vias must disable nearby vertices where vias would conflict (Fig. 2.36a). Installed edges must invalidate all other edges that traverse the same span on a given track (Fig. 2.36b) or, in the case that they are off-grid, be treated as wire-shaped obstruction and checked for intersection. A key difference to obstruction handling is that information on the utilising path is now attached to used edges and vertices. This allows connections from subsequent searches to mutate, for example, an underlying `PolyLine` to receive a via during the export process (and is how we arrive at some of the legalisation problems described above). Notably, because the model must be mutated, installation is the synchronisation bottleneck for multi-threaded path finding.

[11] This is a nice way of putting it. BFG should have a DRC checker, but instead this is implemented by `RoutingGrid::ApplyDumbHackToPatchNearbyVerticesOnSameNetButDifferentLayer` in `routing/routing_grid.cc`.



(a) Using an on-grid vertex invalidates all 8 of its neighbours



(b) Using an edge must cancel all other edges traversing its span

Figure 2.36: Bookkeeping required when routing vertices and edges are used

The final step is to export all routes as wire shapes. `RoutingPaths` are turned into sequences of `PolyLine` wire paths that specify the contiguous components of the wire on each layer, as well as the intervening vias. These are then inflated (Sec. 2.3.3 into complete wire layouts and returned to the user. Figure 2.37 shows the a final layout as exported by the router with some of the aforementioned behaviours annotated. The positions of routing vertices are drawn in with small squares.

2.7.2 Usage

As we did for developing circuits (Sec. 2.6) and layout (Sec. 2.6), we will demonstrate how a designer might use the BFG router to add wiring to their designs by stepping through an example, this time from the `LutB` generator (Sec. 4.4). Refer to `tiles/lut_b.cc` in the BFG repository for the full source.

The designer starts by creating a `RoutingGrid` object with a handle to the active `PhysicalPropertiesDatabase`,

```
RoutingGrid routing_grid(design_db_->physical_db());
```

and then specifying the routing grid parameters: area, pitch, offset and layers (including the directionality of those layers):

```
void LutB::ConfigureRoutingGrid(
    RoutingGrid *routing_grid, Layout *layout) const {
  const PhysicalPropertiesDatabase &db = design_db_->physical_db();
  geometry::Rectangle pre_route_bounds = layout->GetBoundingBox();
```

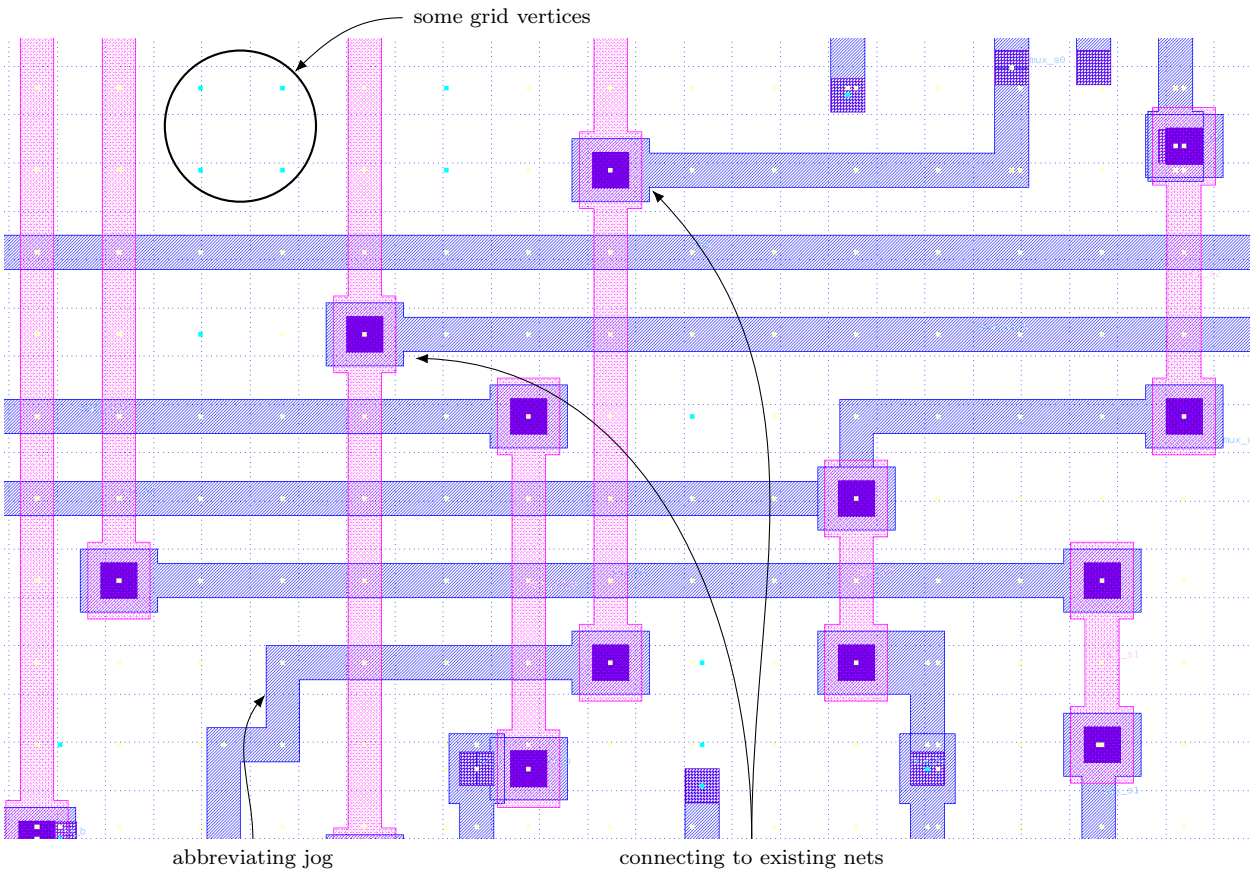


Figure 2.37: Routed wires exported as layout

```
RoutingLayerInfo met1_layer_info =
    db.GetRoutingLayerInfoOrDie("met1.drawing");
met1_layer_info.set_direction(RoutingTrackDirection::kTrackHorizontal);
met1_layer_info.set_area(pre_route_bounds);

RoutingLayerInfo met2_layer_info =
    db.GetRoutingLayerInfoOrDie("met2.drawing");
met2_layer_info.set_direction(RoutingTrackDirection::kTrackVertical);
met2_layer_info.set_area(pre_route_bounds);
```

If the wire width and pitch are not specified, defaults are inferred from the physical database given at construction. The offset can be set through automatic alignment to a known point:

```
auto alignment_point = layout->GetPoint("grid_alignment_point");
if (alignment_point) {  // Fails if no point was found.
  RoutingGridGeometry::AlignRoutingLayerInfos(
```

```
        *alignment_point, &met1_layer_info, &met2_layer_info);
}
```

Next, the designer must set connectivity information for the routing layers they have defined. These could also be inferred from the technology model, but the user may wish to customise the applicable cost functions on a case-by-case basis. For a description of the listed layers and any shorthand notation see Table 1.1 in Section 1.7.5.1. Here we fetch the via layer connecting each of (m_0, m_1), (m_1, m_2) and (m_2, m_3) (respectively v_0, v_1, v_2) and its properties, then override the cost:

```
RoutingViaInfo routing_via_info =
    db.GetRoutingViaInfoOrDie("met1.drawing", "met2.drawing");
routing_via_info.set_cost(0.5);
routing_grid->AddRoutingViaInfo(
    met1_layer_info.layer(), met2_layer_info.layer(), routing_via_info)
    .IgnoreError();

routing_via_info = db.GetRoutingViaInfoOrDie("li.drawing", "met1.drawing");
routing_via_info.set_cost(0.5);
routing_grid->AddRoutingViaInfo(
    met1_layer_info.layer(), db.GetLayer("li.drawing"), routing_via_info)
    .IgnoreError();

routing_via_info = db.GetRoutingViaInfoOrDie("met2.drawing", "met3.drawing");
routing_via_info.set_cost(0.5);
routing_grid->AddRoutingViaInfo(
    db.GetLayer("met3.drawing"), met2_layer_info.layer(), routing_via_info)
    .IgnoreError();

routing_grid->AddRoutingLayerInfo(met1_layer_info).IgnoreError();
routing_grid->AddRoutingLayerInfo(met2_layer_info).IgnoreError();
```

Even though the `RoutingGrid` can only route on metal layers 1 and 2, we still detail the layer immediately above and below so that it can negotiate access to pins on those layers. The user then generates the final grid by issuing:

```
routing_grid->ConnectLayers(met1_layer_info.layer(), met2_layer_info.layer())
    .IgnoreError();
```

Once the grid is created, permanent obstructions can be collected from the `layout` and issued to the router,

```
{
  // Add blockages for all (non-connectable) existing shapes on met1.
  geometry::ShapeCollection shapes;
  layout->CopyNonConnectableShapesOnLayer(
      db.GetLayer("met1.drawing"), &shapes);
  routing_grid->AddBlockages(shapes);
```

```
  }
  {
    geometry::ShapeCollection shapes;
    layout->CopyNonConnectableShapesOnLayer(
        db.GetLayer("met2.drawing"), &shapes);
    routing_grid->AddBlockages(shapes);
  }
```

along with any global signals that should be propagated to the final wire layout:

```
  routing_grid->AddGlobalNet("CLK");
}   // End of LutB::ConfigureRoutingGrid.
```

The designer can then use the `routing_grid` object to connect ports of layout instances or other structures. The most basic mode is to issue calls to `RoutingGrid::AddRouteBetween` with a source and destination point, any additional shapes to avoid, and a set of `EquivalentNets` that are permissible to reuse (these will also become the label for the resulting wire). A common pattern that emerges, however, is that a designer will want to connect a multi-point wire between one set of instance ports while avoiding most others, followed by another multi-point wire between another set of ports, and then another, and so on. We know from the literature that the optimal solution to the multi-point path problem is a minimum rectilinear Steiner tree [89], but BFG does not implement those (yet). Instead, users must route point-to-point between one pair of the route terminals, and then route from each of the next terminals to the established net of the first wire. This is done by successive calls to `RoutingGrid::AddRouteToNet` after the first call to `AddRouteBetween`.

Finding instance ports, excluding and then assembling the paths is a repetitive chore. BFG provides the `RouteManager` class, a convenient wrapper around `RoutingGrid`, to manage these tasks for a collection of routes. It also demonstrates a first step toward implementing more sophisticated optimisation loop within BFG. The user issues route requests as in the basic modality, but with `RouteManager` they are ordered and can even be batched for parallel execution. Returning to our `LutB` example,

```
  RouteManager route_manager(layout, &routing_grid);
  // Blockages on the "li.pin" layer will be automatically cancelled
  // and recreated between routes.
  route_manager.set_auto_cancel_blockages(true);
  route_manager.auto_cancel_layers().push_back("li.pin");
```

sets up the `RouteManager`. Then the user can submit multiple route requests with ease:

```
void LutB::RouteScanChain(
    RouteManager *route_manager,
    Circuit *circuit,
    Layout *layout) {
  /* Omitted: some preamble. */
  for (size_t i = 0; i < scan_order_.size() - 1; ++i) {
```

```
    geometry::Instance *source = scan_order_[i];
    geometry::Instance *sink = scan_order_[i + 1];

    // Connect "Q" from the last memory to "D" of the next one.
    std::set<geometry::Port*> start_ports = source->GetInstancePorts("Q");
    std::set<geometry::Port*> end_ports = sink->GetInstancePorts("D");

    // Decide what set of names the wires can take.
    EquivalentNets net_names;
    for (geometry::Port *port : start_ports) {
      net_names.Add(port->net());
    }
    for (geometry::Port *port : end_ports) {
      net_names.Add(port->net());
    }

    /* Omitted: other circuit-related code. */

    route_manager->ConnectMultiplePorts(
        std::vector<std::set<geometry::Port*>>({
            start_ports, end_ports}),
        net_names).IgnoreError();
  }

  auto result = route_manager->Solve();
  /* Omitted: error handling. */
}
```

After every necessary route has been added to the router, the resulting paths are inflated to size and exported to a standalone `Layout` object. This can then be merged in with the existing design:

```
  Layout *grid_layout = routing_grid.GenerateLayout();
  layout->AddLayout(*grid_layout);
  delete grid_layout;
```

2.8 Ecosystem Integration

In order to be a *useful* tool, BFG must be able to exchange design data with other tools in the VLSI ecosystem[12]. The de-facto standard data interface to each of these tools is usually one or more proprietary file formats, so in order to use third-party tools for LVS, DRC, SPICE and the like, BFG must support reading and writing technology, circuit, layout and simulation information in a host of ways. Since most of these tools are proprietary and closed-source there is simply no other mechanism available. Manufacturing foundries, too,

[12]We call it an "ecosystem" because we are feeling uncommonly charitable.

provide PDK information in a slew of such formats and will only accept design blueprints in the same way.

What makes this unusually challenging is that these formats have very little in common in the way of design ethos, interoperability and/or library support. Circuit topology is commonly represented in a SPICE-format file, except that there are several *dialects* and they are not all inter-compatible. The original SPICE format is a simple declarative text-based format — intended to be written by humans — for specifying components and their connections. Today it has mutated into incompatible variations for each of the major tools: CDL for Cadence Spectre, HSpice by Synopsys, Xyce's format [115], ngspice's format [101], and so on. At a cursory glance each of these is recognisable as SPICE, but they differ in how (if) expressions are supported, delimiting characters, how nested signals are accessed, directives for simulation, how and what analysis functions can be called, and so on. Now it is necessary to support each dialect explicitly and independently.

Then there are the more esoteric formats. The Library Exchange Format (LEF) and Design Exchange Format (DEF) are text-based formats for communicating abstracted views of a design, principally to place-and-route tools (Sec. 1.3.2). *Liberty* files are typically used to describe timing and power summaries for a design. They are text-based but support a brace-delimited nested message structure. Worst of all is Graphic Data System II (GDSII), a format invented by Calma Computer in 1978 for the (then-)efficent manipulation of large layouts. GDSII is a binary format that even features its own binary representations for floating point values, and it is somehow the industry standard for exchanging layout data. OpenAccess[13], a new format which tried to address this, requires a hefty monetary annual subscription fee as part of a mandatory membership of the SI2 consortium [92].

Adding direct support for these formats is a significant barrier to developing open-source EDA software. Writing software to support any single one of these formats is non-trivial, let alone writing software to support all of them. Where manuals documenting each *exist* and can be accessed (they are generally not free *and* open-source), their specifications are lengthy and complex. They require distinct lexers, parsers, decoders and encoders. Though there have been attempts at building FOSS libraries for each format, these typically support a single programming language and their adoption, already inchoate, languishes. As a consequence it is difficult to thoroughly test each library and improve its robustness or usability.

Outside of chip design, at least one software pattern has been developed to successfully solve many of these problems. A contemporary approach within distributed systems is to define *language-agnostic schemas* for the information being exchanged, together with automatic code generators that build language-specific libraries for converting to and from a common binary format implementing each schema. Converting from in-memory to "wire"-level representations (*serialisation*) is a hardened function of the code generators; producers and consumers of the data can write code in whatever language is most appropriate for their task and delegate the logistics of reading and writing to the generated code. Prominent

[13] An oxymoron.

examples of such systems are Protocol Buffers [36], Thrift [94] and Ion [2]. As is common in distributed systems, the message interface becomes the abstraction which enables many independent software components to interact and, as a result, each component can be more specialised.

BFG opts for the modern paradigm; it does not support reading or writing any of the classic formats directly. BFG reads and writes language-agnostic, schema-driven formats for circuit topology, simulation specification, and layout data through a middleware layer called VLSIR. A series of tools written by other researchers then convert between the legacy formats and the VLSIR messages types on our behalf.

2.8.1 VLSIR

VLSIR is a middleware layer for the interchange of IC design data. It was developed in collaboration with Dan Fritchman and Aviral Pandey and is described in more depth in [30]. The name "VLSIR" is a portmanteau of *VLSI* and *IR*, or Intermediate Representation: VLSIR defines language-agnostic schemas for structured process technology, circuit, simulation and layout information. We use Protocol Buffers [36] to define types in the schema that then map to an internal type within each client tool. For example, the VLSIR `vlsir.raw.Layout` type corresponds to the BFG's `Layout` type (Sec. 2.2.3) and is described by the (simplified) snippet of Listing 2.18; the BFG `layout::Instance` type corresponds to the VLSIR `vlsir.raw.Instance` type, shown in Listing 2.19.

Listing 2.18: Snippet of the VLSIR `vlsir.raw.Layout` message

```
message LayerShapes {
  Layer layer = 1;
  repeated Rectangle rectangles = 2;
  repeated Polygon polygons = 3;
  repeated Path paths = 4;
}

message Layout {
  // Cell Name
  string name = 1;
  // Shapes, grouped by layer
  repeated LayerShapes shapes = 2;
  // Instances of other cells
  repeated Instance instances = 3;
  // Text Annotations
  repeated TextElement annotations = 4;
}
```

Listing 2.19: Snippet of the VLSIR `vlsir.raw.Instance` message

```
message Instance {
  // Instance Name
  string name = 1;
  // Cell Reference
  vlsir.utils.Reference cell = 3;

  // Location of the defined Cell's origin
```

```
  // this location holds regardless of reflection or rotation
  Point origin_location = 4;
  // Vertical reflection about x-axis,
  // applied *before* rotation.
  bool reflect_vert = 6;
  // Angle of rotation (degrees),
  // Clockwise and applied *after* reflection
  int32 rotation_clockwise_degrees = 7;
}
```

BFG is responsible for packing and unpacking VLSIR types, which it does through instance methods with names starting `ToVLSIR` and `FromVLSIR`. Protocol Buffers then serialise and deserialise these types to and from either a binary format (for efficient transmission and storage) or a text format (for readability by humans). The text format is useful for specifying static data. For example, we specify the description of layers available in the SkyWater 130 nm technology to BFG using a text-format `vlsir.Technology` message. A snippet of this file is shown in Listing 2.20, and the corresponding schema is shown in Listing 2.21.

BFG relies on other tools in the VLSIR ecosystem to convert VLSIR messages into necessary formats. The `proto2gds` tool, part of the Layout21 [30] project, converts VLSIR layout messages into GDSII files. To import existing layout files, we use the `gds2proto` tool which is also part of the same library. VLSIR's utility library provides methods to turn VLSIR circuit messages into various SPICE dialects (methods which came from work on the Hdl21 library [30]).

VLSIR reduces the barrier to interfacing with the EDA ecosystem. Tools like BFG need not worry about repeating the work to read and write various convoluted legacy formats; data interchange uses consistent and well-defined patterns. What's more, each tool can be written in a different language. At the time of writing, Protocol Buffers had robust support for 17: Layout21, `proto2gds` and the like are written in Rust, the Hdl21 SPICE converters are written in Python, and BFG is written in C++. Suddenly we are free to implement an idea or tool in whatever language we are most comfortable or that we believe is most suitable.

Though tools that support VLSIR can be as specialised or as general-purpose as their authors see fit, we believe it should be used to apply the Unix philosophy to EDA tooling. As Doug McIlroy puts it in [85]:

> This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

We have substituted VLSIR for text streams as the universal interface. `proto2gds` and `gds2proto` perform a single function well.

A notable weakness of using structured serialisation formats like Protocol Buffers is the cost of serialisation and deserialisation itself. EDA tools typically convert input formats into their own proprietary internal databases, which are intended to have efficient in-memory representations so that they can be quickly manipulated when the designs are large (that is, multiple gigabytes). The usual trade-off between in-memory and on-disk efficiency applies:

Listing (2.20) Snippet of the VLSIR `vlsir.Technology` message for SkyWater 130 nm, from `sky130.technology.pb.txt`

```
name: "Sky130"
# met1.drawing, where most shapes go, is
# identified by the integer pair 68/20 in GDS
layers {
  name: "met1"
  purpose {
    description: "drawing"
    type: DRAWING
  }
  index {
    major: 68
    minor: 20
  }
}
# met1.pin defines pins for met1.drawing, itself
# identified by 68/16
layers {
  name: "met1"
  purpose {
    description: "pin"
    type: PIN
  }
  index {
    major: 68
    minor: 16
  }
  pin_access_for {
    major: 68
    minor: 20
  }
}
# met1.label is the label for met1.drawing,
# itself identified by 68/5
layers {
  name: "met1"
  purpose {
    description: "label"
    type: LABEL
  }
  index {
    major: 68
    minor: 5
  }
  labels {
    major: 68
    minor: 16
  }
  labels {
    major: 68
    minor: 20
  }
}
```

Listing (2.21) Snippet of the definition of VLSIR's `vlsir.Technology` message

```
package vlsir.tech;
message Technology {
  string name = 1;
  repeated Package packages = 11;
  repeated LayerInfo layers = 101;
}

enum LayerPurposeType {
  UNKNOWN = 0;
  LABEL = 1;
  DRAWING = 2;
  PIN = 3;
  OBSTRUCTION = 4;
  OUTLINE = 5;
}

message LayerPurpose {
  // Short-hand description of the purpose of the
  // sub-layer, e.g. "drawing".
  string description = 1;
  LayerPurposeType type  = 2;
}
message LayerInfo {
  // A canonical shorthand name for the layer,
  // e.g. "met1".
  string name = 1;

  // Here you might put other variations of the
  // layer name for different programs, e.g.
  // whatever magic should call the layer.

  LayerPurpose purpose = 11;
  // An integer index identifying the major layer
       .
  uint64 index = 21;
  // An integer index identifying the sub layer,
  // or purpose.
  uint64 sub_index = 31;
}
```

in memory data is best aligned with cache line size and grouped with the working set, while on-disk the data should be as compressed as possible. Alternative frameworks exist to address this shortcoming. Both FlatBuffers [34] and Cap'n'Proto [105], for example, match the wire-format of a message to its in-memory layout for trivial serialisation, but do so at the expense of larger message sizes. If this proves to a be significant problem, "VLSIR 2" can use a more appropriate underlying framework.

2.8.2 BigSpicy

BigSpicy [82] is our Python tool for manipulating circuit netlists in various formats. It was born of the need to merge structural Verilog, SPICE and parasitic extracted netlists in SPEF format, and is the FOSS equivalent of the commercial `v2lvs` tool. BigSpicy can read and write VLSIR circuit topologies, so once data is imported and combined it can be manipulated and then passed to other VLSIR tools. BigSpicy can also generate custom analyses: it provides a convenient circuit model with which Python scripts can easily traverse the circuit graph. As an example, one of its functions is to identify all of the sub-graphs in a circuit that correspond to parasitic-extracted *wires*. (These are the interiors of graph regions enclosed by devices.) The transient response of each individual wire can then be simulated, allowing for the development of wire delay and power models that, one day, can be used to configure the BFG router.

2.9 Microservices for VLSI

A contemporary idea for the construction of scalable distributed software systems is to to use *microservices* [107]. These are independent executables, running on networked machines, that play a specific role or perform a component function within a larger software process. Microservices typically expose functionality through Remote Procedure Calls (RPCs), and have been the foundation of large-scale web services built in the last two decades.

The VLSI design flow (Sec. 1.3.2) comprises multiple computationally-intensive steps that block the development pipeline until their completion. They also tend to consume significant system resources (processor time and memory) while doing so, making it hard to do much else. The contemporary paradigm for chip designers is to invest in multiple powerful computers and to then load balance users' tasks by wholesale redirecting their entire flow to machine under the least load. This is cumbersome in practice because most work is done through remote terminals (such as with VNC, a popular remote desktop client). Not only is the user experience slow, but the machines become responsible for *all* of a user's work, not just the resource-intensive elements. Distributed software design solves this by specialising machines for roles and committing their resources to those roles alone. The user experience is masked by front-end software that asynchronously calls on each service to perform some task in the background, reporting the results as required, and freeing the user to continue their other work.

```
{
  "predefined_technology":
      "TECHNOLOGY_SKY130",
  "grid_definition": {
    "layers": [
      {
        "name": "met1.drawing",
        "direction":
            "TRACK_DIRECTION_HORIZONTAL",
        "area": {
          "lower_left": { "x": 0, "y": 0 },
          "upper_right": {
            "x": 1000,
            "y": 1000
          }
        },
        "offset": 330
      },
      ...
    ],
    "vias": [
      {
        "between_layer": "li.drawing",
        "and_layer": "met1.drawing",
        "cost": 0.5
      },
      ...
    ]
  }
}
```

(a) `setup.json`

```
      {
  "grid_id": 1,
  "net_route_orders": [
    {
      "net": "A",
      "points": [
        {
          "point": { "x": 10, "y": 10 },
          "layer_name": "met1.pin"
        },
        {
          "point": { "x": 100, "y": 100 },
          "layer_name": "met1.pin"
        },
        {
          "point": { "x": 300, "y": 50 },
          "layer_name": "li.pin"
        },
        {
          "point": { "x": 900, "y": 900 },
          "layer_name": "li.pin"
        }
      ]
    }
  ]
}
```

(b) `request.json`

BFG explores bringing microservice architecture to bear on AMS VLSI design. In particular, routing and simulation are two computationally intensive tasks that we consider to be in the "core loop" of the AMS design process. The BFG router can be run as a microservice. Separately, we have developed `spice_server`, a microservice wrapper for common SPICE simulators. Both use the Google gRPC library [37], in conjunction with Protocol Buffers [36] to expose key operations. The router service, for example, exposes the same API as `RoutingGrid` and `RouteManager`. Designers using BFG can call on these services to perform routing and run simulations in the background on more powerful hardware elsewhere. If excess resources are available, they could even issue multiple background requests to experiment with different design elements, run routing and simulate the circuit performance of the results.

A key benefit of microservice architecture is that services are agnostic to the form of the client that invokes them. They can be used by any AMS framework, in one of many languages, by scripts, or from the command line. For example, after starting up the server from

the command line, the user can issue a command-line gRPC call, with a request expressed in common Javascript Object Notation (JSON), to configure it in a similar way to what we saw in Section 2.7.2:

```
cat setup.json | ~/go/bin/grpcurl -plaintext -d @ \
    localhost:8222 bfg.router_service.RouterService/CreateRoutingGrid
```

where the contents of `setup.json` are shown in 2.39a. The user can then issue route requests in the same way,

```
cat request.json | ~/go/bin/grpcurl -plaintext -d @ \
    localhost:8222 bfg.router_service.RouterService/AddRoutes
```

where `request.json` is shown in Figure 2.39b, and receive the response directly:

```
{
  "status": {},
  "routes": [
    {
      "net": "A",
      "points": [
        {
          "x": "10",
          "y": "10"
        },
        {
          "x": "50",
          "y": "10"
        },
        {
          "x": "50",
          "y": "100"
        },
        {
          "x": "100",
          "y": "100"
        }
      ],
      "layers": [
        "met1.drawing",
        "met1.drawing",
        "met1.drawing"
      ]
    }
  ]
}
```

The `spice_server` [83] works in a similar way. The server must be configured at start-up with the path to one or more SPICE server executables. Once it is running, users make

RPC requests to run simulations with a particular simulator. They can either providing a directory of SPICE files to copy over and have executed, or expressing the circuit data and simulation testbench using the VLSIR schema. The simulation data can be returned as the verbatim file output or corralled into the VLSIR schema again. Hdl21 [30] implements a similar mechanism for wrapping calls to a SPICE server and provides a good example of an alternate implementation.

Chapter 3

Interconnect

An overarching aim of this work is to close the performance gap between academic and commercial fabrics (Sec. 1.6). We know that contemporary commercial FPGA architectures have diverged significantly from established literature (Sec. 1.5), and commercial device vendors do not share the circuit- or physical-level designs of their devices. In order to use BFG to make a "modern" fabric, it is necessary to first understand how they are built in some detail. The Xilinx UltraScale [3] is a market-leading FPGA device [76] that serves as our example of the state of the art. In this chapter, we reverse-engineer the interconnect architecture of the UltraScale from publicly available data. We determine the patterns that guide its structure and show how these can be generalised into algorithms for the creation of similar topologies.

3.1 Xilinx UltraScale

The design of the interconnect in the Xilinx UltraScale family of FPGAs is a significant departure from the traditional academic approaches of Section 1.4. The details of this architecture are proprietary and have not been published in meaningful detail. However, using published user manuals and publicly available data, we were able to reverse-engineer a typical UltraScale topology and reason about the physical design considerations that likely informed its organisation. In the remainder of this chapter, we explain the methodology by which we studied its architecture and show how it differs to accepted practice. In Section 4.10 we attempt to generalise these insights into a procedural generator for UltraScale-like fabrics.

The Xilinx UltraScale is arranged as follows. As we have seen (Sec. 1.4), the workhorses of any FPGA are the LUTs which implement arbitrary boolean functions and which, when combined with a register, can (in principle) implement any digital design. Elsewhere in this dissertation we call this combination a Basic Logic Element (BLE); Xilinx calls this a Configurable Logic Block (CLB). The UltraScale CLB, whose logical view is summarised in Figure 3.1, comprises a 6-LUT which can be fractured into two 5-LUTs, two bypass inputs

(X, I), and two registers (Q1, Q2). The registers are clocked by the application clock, whose connections are not shown. Each register's input can be programmably taken from either LUT output or either bypass input. Outputs are available from each fractured LUT, the bypass inputs and the registers. We will use the terms "CLB" and "BLE" interchangeably.

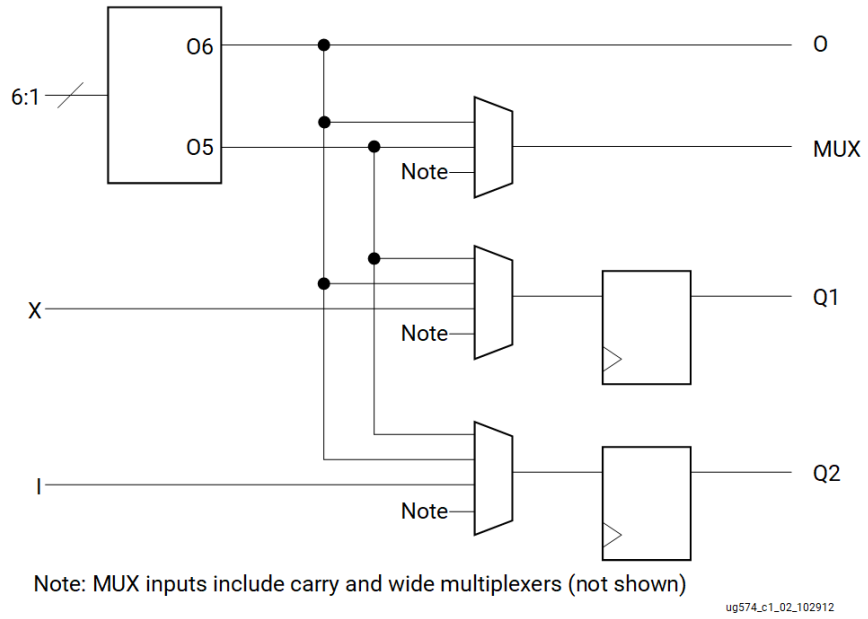


Figure 3.1: The Basic Logic Element of a Xilinx UltraScale is its Configurable Logic Block [4]

Eight BLEs are grouped into a "slice", which includes the interconnect wiring to connect them to each other and to the rest of the fabric. A "tile" contains two horizontally-adjacent "slices", one west and one east. Some slices can be configured as LUT RAM, where the LUT memories can be written by user signals at runtime. Unlike in the traditionally-distinct BLEs and connection boxes, all switching in the UltraScale is performed within these tiles: inter-tile wires are driven directly from one tile and terminate in another, where they can be used as inputs to the local slices or can be connected to an outgoing wire. There are no intervening switch boxes. Any time a path needs to join two long wires or turn a corner, it does so through a tile. All of the wiring is uni-directional, which we know from [60] to be a good idea.

There is evidence to suggest that applications implemented on FPGAs commonly exhibit columnar stratification within their logic. In [118], Young shows that the logical representation of digital designs can commonly be arranged into layers of boolean functions. Data, which are typically several bits wide, move through layers of functions together, carving a path that is wide but many layers deep. Feedback paths are usually present but far fewer in number than the forward-biased data flow paths. Recalling from Section 1.4 that FPGAs excel at exploiting spacial parallelism, the claim is not that all designs demonstrate

this property, but rather that designs intended for FPGAs really ought to do so in order to maximise performance.

It is therefore unsurprising to learn that the UltraScale architecture explicitly uses a columnar arrangement of logic elements (Fig. 3.2). Slices in the same tile are connected by short, direct connections through a shared switching matrix. The tiles are repeated vertically to form a column. Within a column, additional wiring directly connects vertically-adjacent slices, allowing them to cooperate to implement the functionality of one of Young's layers.

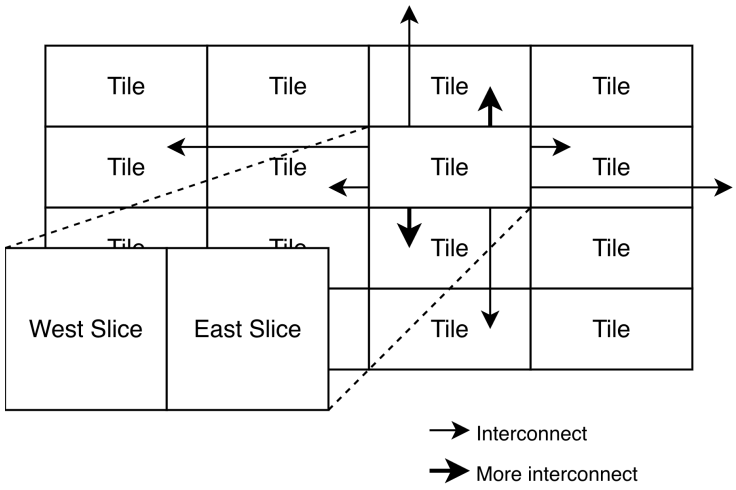


Figure 3.2: Slices and tiles in the UltraScale fabric

The interconnect topology within each slice of a tile (east, west) is broadly symmetric. As summarised in Figure 3.3, it can be subdivided into three major regions: (1) incoming wiring is distributed to the inputs of Logic Elements through an Input Interconnection Block (IIB, see Section 3.4); (2) the outputs of Logic Elements are distributed to outgoing wiring through an Output Interconnection Block (OIB, see Section 3.5); and (3) an interconnect crossbar directly connects incoming wiring to longer outgoing wiring (Section 3.6). The IIB and OIB contain feedback paths to other regions within the slice and shortcut paths to neighbours.

3.2 Methodology

The RapidWright tool [59] was created to allow researchers to fine tune circuit designs implemented for Xilinx FPGAs. The authors describe it as "an open source Java framework that enables netlist and implementation manipulation of modern Xilinx FPGA and SoC designs" [111]. It provides a graph-based model of all of the components in a fabric for a specific device, and includes such models for a multitude of commercially available UltraScale

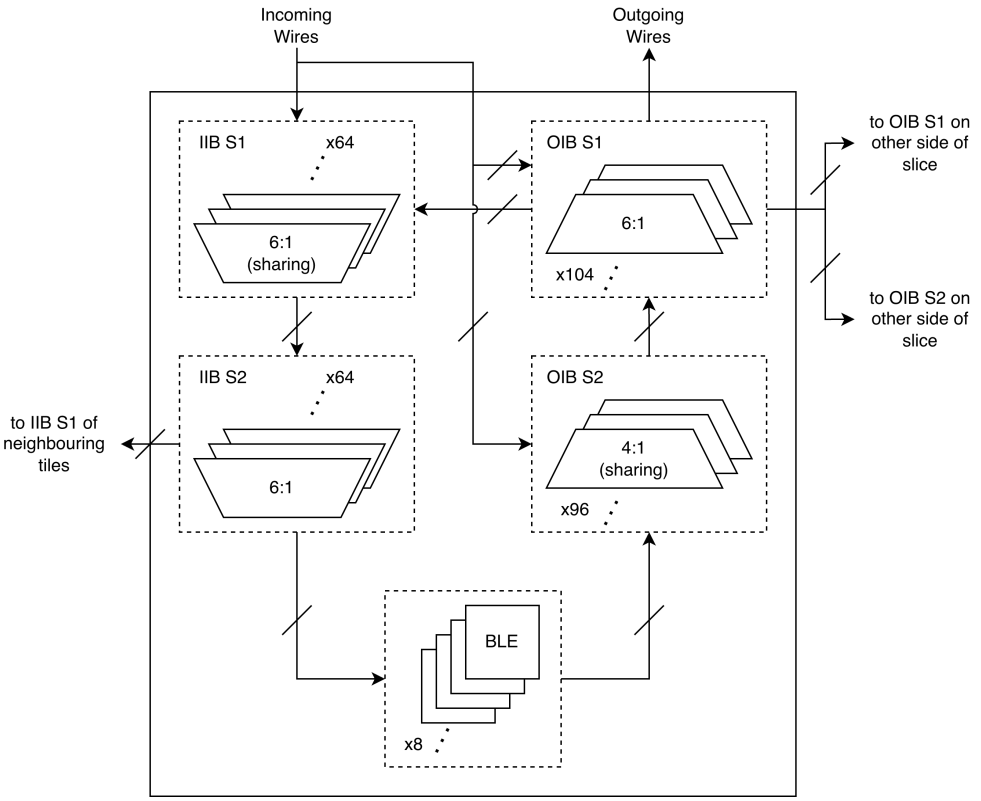


Figure 3.3: Overview of the interconnect topology within an UltraScale Slice

devices. Through a Java API, the relationship between architectural entities can be queried. The same can be used to mutate implemented designs.

Part of the RapidWright model for each device is a description of the interconnect wiring. The RapidWright documentation provides Figure 3.4 to explain its model of a fabric, where we see two interconnect sites (`INT`), a regular LUT slice (`CLEL`) and a LUT RAM slice (`CLE_M`). Noting how the `CLE_M` site's x-coordinate matches its right-hand interconnect tile, and likewise how the `CLEL_R` site's x-coordinate matches its left-hand interconnect tile, we can immediately conclude that the `CLE` sites correspond to *slices*. A left-hand slice, interconnect block and right-hand slice together form a *tile*. All sites within a tile share the same coordinate. The labelled wire is what RapidWright terms a *node*, "a collection of electrically connected wires that spans one or more tiles" [111].

Listing 3.1: RapidWright program to extract interconnect nodes, courtesy of Jonathan Greene and Eddie Hung. © 2023 AMD; Apache 2.0 license.

```
package com.xilinx.rapidwright.examples;

import com.xilinx.rapidwright.device.Device;
import com.xilinx.rapidwright.device.IntentCode;
import com.xilinx.rapidwright.device.Node;
```

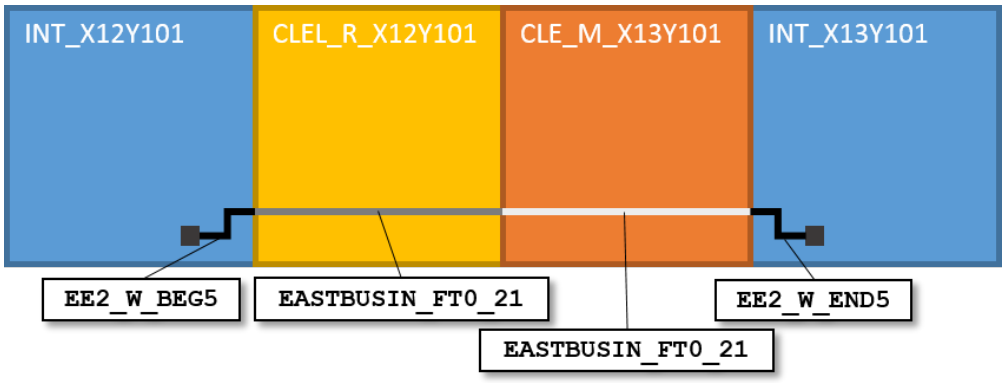


Figure 3.4: Nodes and their names in the RapidWright model

```
import com.xilinx.rapidwright.device.PIP;
import com.xilinx.rapidwright.device.SitePin;
import com.xilinx.rapidwright.device.Tile;
import com.xilinx.rapidwright.device.TileTypeEnum;
import com.xilinx.rapidwright.util.Utils;

public class Connectivity {
  public static String getTypeName(Node node) {
    return node.getIntentCode().toString();
  }

  public static String getInstName(Node node) {
    return node.getTile().getTileTypeEnum() + ''/'' + node.getWireName();
  }

  public static void main(String args[]) {
    Device device = Device.getDevice(''xczu3eg-sbva484'');
    Tile tile = device.getTile(''INT_X20Y100'');

    float delay = 0;

    for (PIP pip : tile.getPIPs()) {
      Node start = pip.getStartNode();
      Node end = pip.getEndNode();
      assert(end.getTile() == tile);

      if (start.getTile().getTileTypeEnum() != TileTypeEnum.INT) {
        if (start.getTile().getName().startsWith(''RCLK_'')) {
          // Ignore clocking
          continue;
        }
        assert(Utils.isCLB(start.getTile().getTileTypeEnum()));
      }

      int deltaX = start.getTile().getTileXCoordinate() - end.getTile().getTileXCoordinate();
      int deltaY = start.getTile().getTileYCoordinate() - end.getTile().getTileYCoordinate();

      String comment = '''';
      SitePin sitePin = end.getSitePin();
      if (sitePin != null) {
```

```
        comment = '' # '' + sitePin.getTile().getTileTypeEnum() + ''.'' + sitePin.getPinName();
      }

      System.out.println(''CONN '' + getTypeName(start) + '' '' + getInstName(start) + '' '' +
          getTypeName(end) + '' '' + getInstName(end) + '' '' +
          deltaX + '' '' + deltaY + '' '' + delay + comment);
    }

    // [A-H](X|_I)
    System.out.println(''LOGIC_IN '' + IntentCode.NODE_PINBOUNCE);
    // CKEN[1-4]|SRST[1-2]|[A-H][1-6]
    System.out.println(''LOGIC_IN '' + IntentCode.NODE_PINFEED);
    // [A-H](_O|Q|Q2)
    System.out.println(''LOGIC_OUT '' + IntentCode.NODE_CLE_OUTPUT);
  }
}
```

We used a RapidWright program to extract the list of all connections within a typical *interconnect* site (`INT_X20Y100`) in a large UltraScale device (`xczu3eg`), which represents the entire interconnect for a single tile. (The program, courtesy of Jonathan Greene and Eddie Hung, is shown in Listing 3.1.) We chose this particular tile to be somewhere in the middle of the fabric, in order to avoid the depopulation of tiles near the edges of the fabric, which we expect to have fewer neighbours and therefore fewer connections. Our tile is 20 tiles across in the x-axis and 100 tiles up in the y-axis.

A snippet of the edge list output by the RapidWright program for this tile is shown in Listing 3.2. The key columns (fields) are the source instance (`sourceInst`) and destination instance (`toInst`). The `deltaX` and `deltaY` fields indicate the relative position of the *source* instance, measured in the number of tiles it is offset from the current position. The node type fields, `fromType` and `toType`, describe the type of entity in the RapidWright schema. We found that these provided a useful hint to the kind of wiring being described in the interconnect. The `comment` field, where present, indicates correspondence to entities within the logic slices neighbouring the interconnect tile.

We wrote a tool, `fiddle.py`, to extract the edges going into and out of a particular instance. The tool reads the edge list and accumulates its own internal model of the connectivity graph with all available data for each node. From this model we can produce a simple diagram of a particular node's connectivity and, as will become useful later, an adjacency matrix for all participating nodes. For example, executing the program as follows will generate the simple connectivity diagram shown in Figure 3.5:

```
./fiddle.py --show_node "INT/INT_NODE_SDQ_18_INT_OUT0" \
    | dot -Tpng -o output.png
```

It is also possible to repeatedly specify the `-show_node` argument to query multiple node patterns, which will be useful for finding surprising relationships between nodes. For example, the following command will generate the output of Figure 3.6

```
./fiddle.py \
```

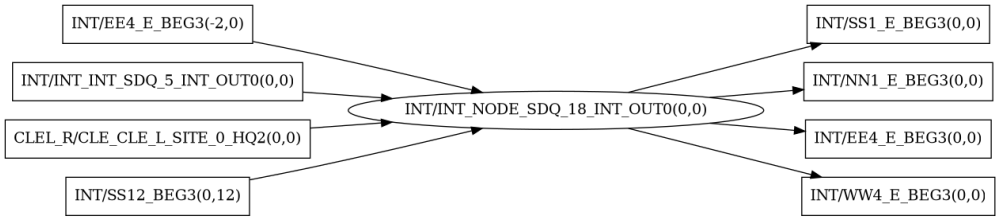


Figure 3.5: Example output of the `fiddle.py` tool

```
--show_node "INT/INODE_E_9_FT1(0,0)" \
--show_node "INT/INT_NODE_IMUX_31_INT_OUT0(0,0)" \
| dot -Tpng -o output.png
```

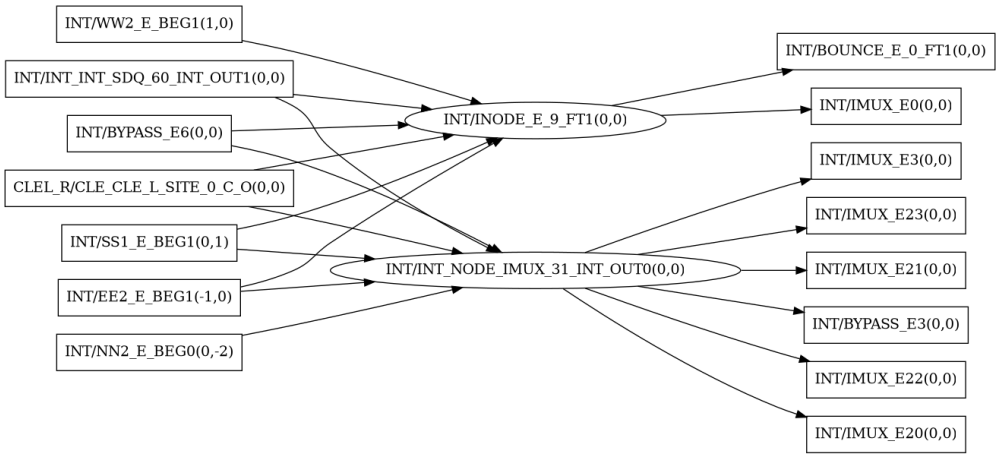


Figure 3.6: Example of compound queries with the `fiddle.py` tool

Listing 3.2: Snippet of the (truncated) X20_Y100 interconnect edge list produced by RapidWright

```
,fromType,fromInst,toType,toInst,deltaX,deltaY,delay,,comment,ble
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_1_INT_OUT0,NODE_PINBOUNCE,INT/BYPASS_E4,0,0,0,#,CLEL_R.A_I,C
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_3_INT_OUT0,NODE_PINBOUNCE,INT/BYPASS_E4,0,0,0,#,CLEL_R.A_I,C
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_30_INT_OUT0,NODE_PINBOUNCE,INT/BYPASS_E4,0,0,0,#,CLEL_R.A_I,
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_5_INT_OUT1,NODE_PINBOUNCE,INT/BYPASS_E4,0,0,0,#,CLEL_R.A_I,C
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_7_INT_OUT1,NODE_PINBOUNCE,INT/BYPASS_E4,0,0,0,#,CLEL_R.A_I,C
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_9_INT_OUT1,NODE_PINBOUNCE,INT/BYPASS_E4,0,0,0,#,CLEL_R.A_I,C
CONN,INTENT_DEFAULT,INT/VCC_WIRE,NODE_PINBOUNCE,INT/BYPASS_E4,0,0,0,#,CLEL_R.A_I,CLEL_R.A
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_12_INT_OUT0,NODE_PINFEED,INT/IMUX_E10,0,0,0,#,CLEL_R.A1,CLEL
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_14_INT_OUT0,NODE_PINFEED,INT/IMUX_E10,0,0,0,#,CLEL_R.A1,CLEL
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_16_INT_OUT0,NODE_PINFEED,INT/IMUX_E10,0,0,0,#,CLEL_R.A1,CLEL
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_18_INT_OUT1,NODE_PINFEED,INT/IMUX_E10,0,0,0,#,CLEL_R.A1,CLEL
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_20_INT_OUT1,NODE_PINFEED,INT/IMUX_E10,0,0,0,#,CLEL_R.A1,CLEL
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_23_INT_OUT1,NODE_PINFEED,INT/IMUX_E10,0,0,0,#,CLEL_R.A1,CLEL
CONN,INTENT_DEFAULT,INT/VCC_WIRE,NODE_PINFEED,INT/IMUX_E10,0,0,0,#,CLEL_R.A1,CLEL_R.A
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_15_INT_OUT1,NODE_PINFEED,INT/IMUX_E8,0,0,0,#,CLEL_R.A2,CLEL_
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_17_INT_OUT1,NODE_PINFEED,INT/IMUX_E8,0,0,0,#,CLEL_R.A2,CLEL_
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_19_INT_OUT1,NODE_PINFEED,INT/IMUX_E8,0,0,0,#,CLEL_R.A2,CLEL_
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_5_INT_OUT0,NODE_PINFEED,INT/IMUX_E8,0,0,0,#,CLEL_R.A2,CLEL_R
CONN,NODE_LOCAL,INT/INT_NODE_IMUX_7_INT_OUT0,NODE_PINFEED,INT/IMUX_E8,0,0,0,#,CLEL_R.A2,CLEL_R
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_68_INT_OUT1,NODE_HQUAD,INT/WW4_W_BEG4,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_69_INT_OUT1,NODE_HQUAD,INT/WW4_W_BEG4,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_73_INT_OUT0,NODE_HQUAD,INT/WW4_W_BEG4,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_74_INT_OUT0,NODE_HQUAD,INT/WW4_W_BEG4,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_76_INT_OUT1,NODE_HQUAD,INT/WW4_W_BEG5,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_77_INT_OUT1,NODE_HQUAD,INT/WW4_W_BEG5,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_79_INT_OUT0,NODE_HQUAD,INT/WW4_W_BEG5,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_80_INT_OUT0,NODE_HQUAD,INT/WW4_W_BEG5,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_82_INT_OUT1,NODE_HQUAD,INT/WW4_W_BEG6,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_83_INT_OUT1,NODE_HQUAD,INT/WW4_W_BEG6,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_86_INT_OUT0,NODE_HQUAD,INT/WW4_W_BEG6,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_87_INT_OUT0,NODE_HQUAD,INT/WW4_W_BEG6,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_89_INT_OUT1,NODE_HQUAD,INT/WW4_W_BEG7,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_90_INT_OUT1,NODE_HQUAD,INT/WW4_W_BEG7,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_93_INT_OUT0,NODE_HQUAD,INT/WW4_W_BEG7,0,0,0,,,
CONN,NODE_LOCAL,INT/INT_NODE_SDQ_94_INT_OUT0,NODE_HQUAD,INT/WW4_W_BEG7,0,0,0,,,
```

Entity names in the RapidWright edge list are often repeated, as would be expected for structures repeated in every tile. To disambiguate them, our inspector appends the relative position of the entity to its name, and to simplify lookup, entities are matched by longest-prefix using a trie. If one does not do this the data become very confusing.

The tool also compiles information on indirectly mentioned entities, such as the BLEs in adjacent logic slices. We know that there are eight BLEs and correspondingly, we find eight objects labelled A-H. Using our tool we can visualise the input connectivity for the first BLE as shown in Figure 3.7. We find the expected LUT inputs (1-6) and both bypass inputs (X, I). We can likewise visualise its outputs (Fig. 3.8) to find each of the corresponding outputs from Figure 3.1: O, MUX, Q1 and Q2.

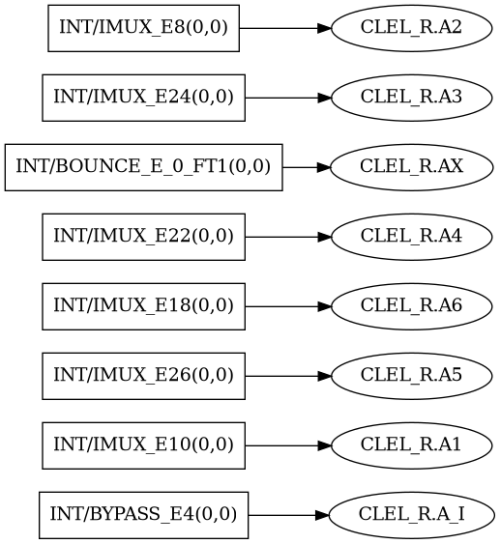


Figure 3.7: Inputs (1-6, X, I) to a BLE in the UltraScale

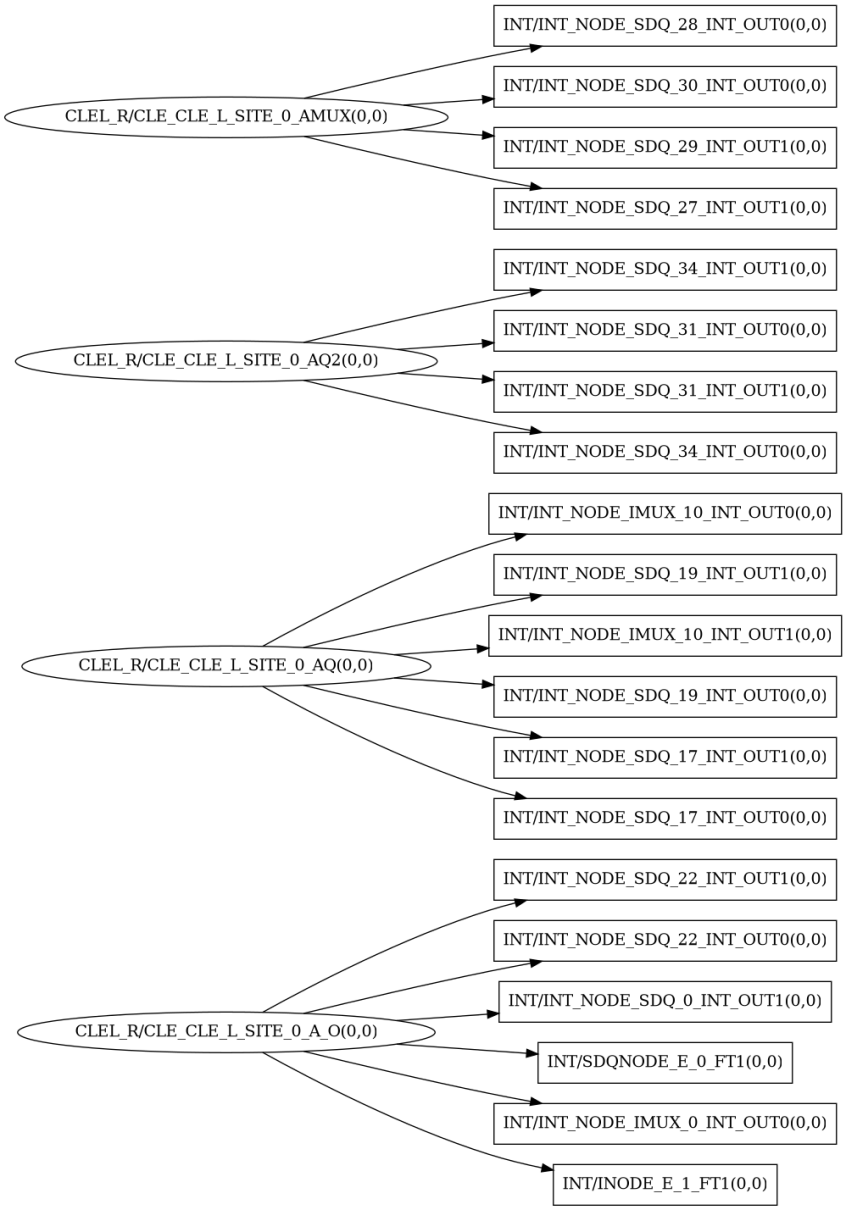


Figure 3.8: Outputs (MUX, O, Q, Q2) from a BLE in the UltraScale

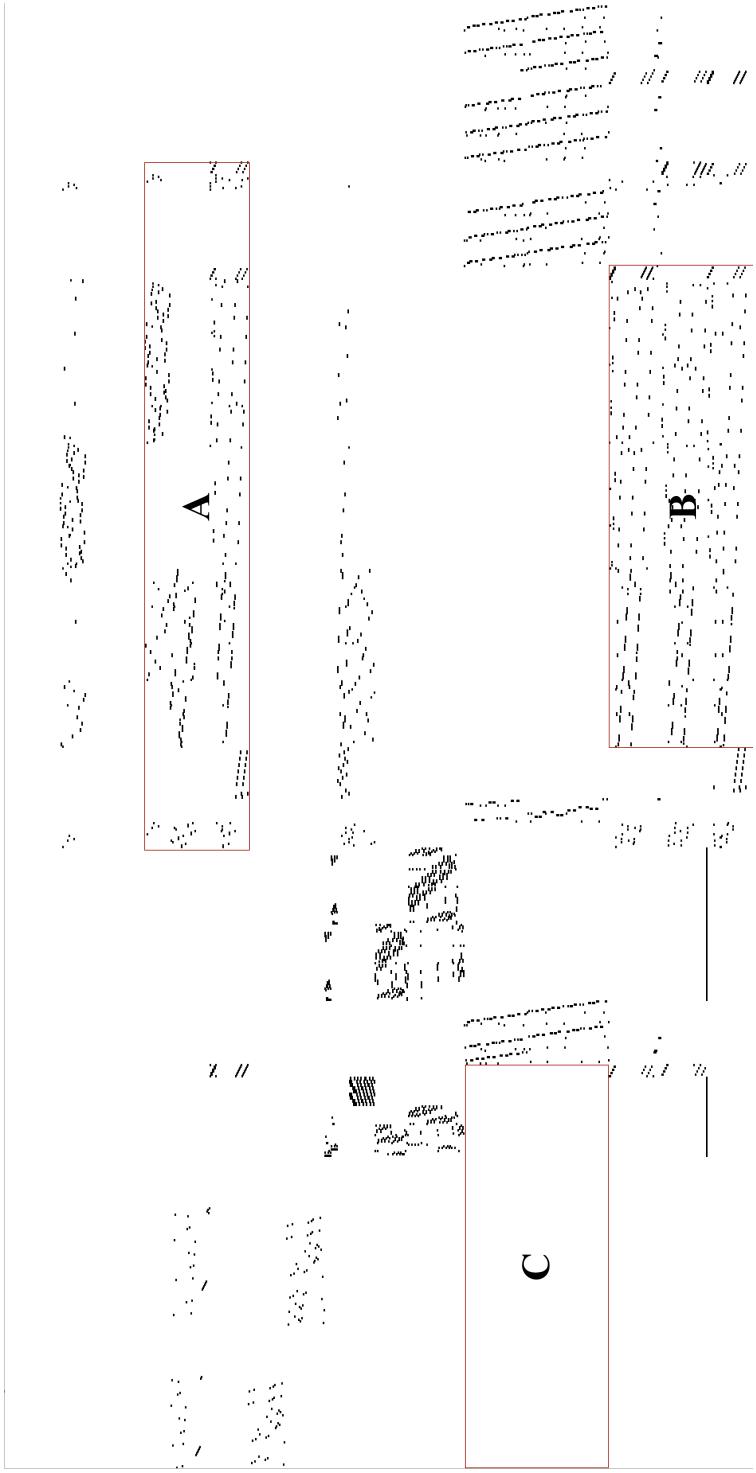


Figure 3.9: UltraScale tile adjacency matrix

To determine the overall structure shown in Figure 3.3, we first tried using the inspector tool `fiddle.py` to produce an adjacency matrix for the entire network, we which have shown in Figure 3.9. It was intractable to reason in detail about this 933×933 table by hand. Nevertheless, as is drawn on Figure 3.9 directly, the overall matrix shows clearly delineated regions of high connectivity (e.g. A, B) and no connectivity (e.g. C). Since the columns and rows in this view are sorted alphabetically, the adjacency matrix also suggested that similarly-named nodes will appear in the same groups. To confirm this, we grouped together nodes with like names and sampled their connectivity with the inspector using diagrams like in Figures **??** and 3.7. Clear clusters emerged, and within each clear sub-clusters emerged as well. It was from these clusters that we were able to identify the broad components of the network as the IIB, OIB, and crossbar sections, as well as their constituent stages (Section 3.1). Table 3.1 summarises how we assigned node names to each cluster.

Table 3.1: UltraScale interconnect entity names and their assigned groups

Node name	Group	Stage	Slice
INODE_{E,W}_x_FTy	IIB	1	East (E) or West (W)
INT_NODE_IMUX_x_INT_OUTy	IIB	1	Shared
BOUNCE_{E,W}_x_FTy	IIB	2	East (E) or West (W)
BYPASS_{E,W}x	IIB	2	East (E) or West (W)
IMUX_{E,W}x	IIB	2	East (E) or West (W)
INT_NODE_SDQ_x_INT_OUTy	OIB	2	Shared
SDQNODE_{E,W}_x_FTy	OIB	1	East (E) or West (W)
{NN,EE,SS,WW}{1,2,4}_{E,W}_BEGx	OIB	1	East (E) or West (W)
{NN,EE,SS,WW}12_BEGx	OIB	1	Shared
CTRL_{E,W}x	Control		East (E) or West (W)
INT_NODE_GLOBAL_x_INT_OUTy	Global		Shared

3.2.1 Physical Hints

Node names embed three interesting details about the physical organisation of the network. First, observe that each node name contains an ordinal number $x \in \mathbb{N}$, which appears to be a straightforward enumeration of all of the entities of a certain type. However, we hypothesise that x also bears a direct relationship to the physical placement of the node in the circuit layout, with the start and end entities being placed near the edges of the block and numerically-nearby entities being placed physically nearby. The principle evidence for this, as we will see in later sections, is that (a) ordinals near the start (number 0) and end of each range tend to be used to connect to neighbouring tiles; and (b) when fanning-out across a cluster, connections are often made to groups of ordinals with nearby ordinals. Under our

hypothesis the former approach results in shorter, faster wiring, and the latter results in reduced routing congestion. It would also be the most straightforward and intuitive way for a human to number things.

Second, we observe that otherwise-like names are sometimes distinguished by a second ordinal $y \in \mathbb{N}$, as for example in the case of `INT_NODE_SDQ_68_OUT0` and `INT_NODE_SDQ_68_OUT1`. We used our inspector to query nodes that differ by only the final ordinal y, and each time we found that the two nodes shared all but one input, as in Figure 3.10.

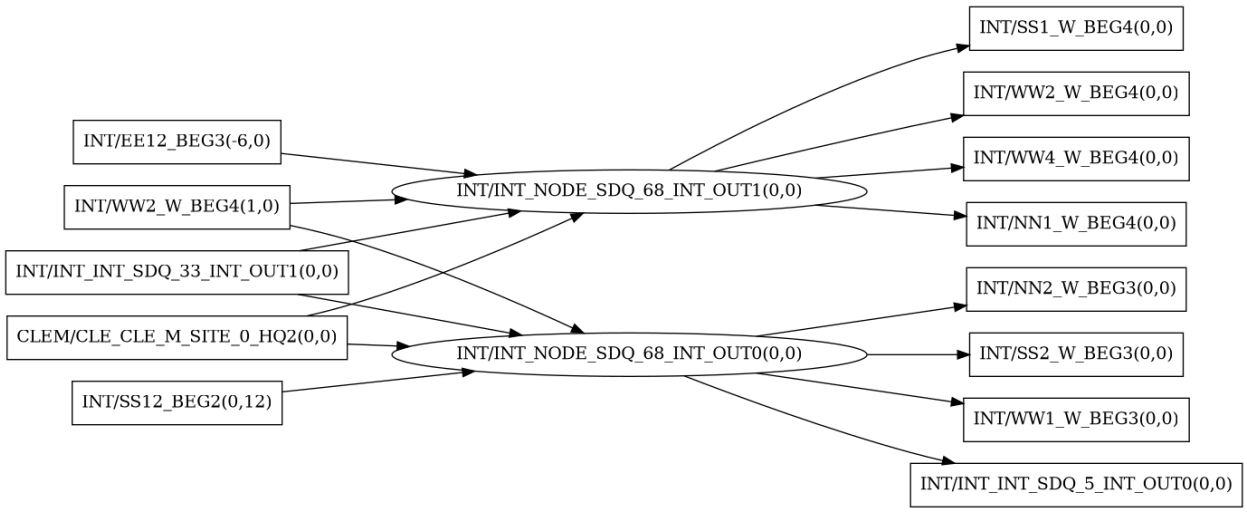


Figure 3.10: Input sharing across entities distinguished by the second ordinal

This logical arrangement coincides with an efficient way of laying out transmission-gate multiplexers. In the case where a one-hot encoding scheme is used *and* there are exactly two independent outputs, the transmission gates for each of two multiplexers can be interleaved so that they lie back-to-back, as for example in Figure 3.12. The corresponding circuit is shown in Figure 3.11. Each gate shares an input node on one side and an output node on the other; the one-hot encoding scheme ensures that no input can be inadvertently driven by another input. An interesting parity emerges: as additional gates are stacked, the two multiplexers alternate between sharing all inputs and sharing all but one. In Figure 3.12 two 3:1 multiplexers share all 3 inputs. An additional transmission gate at each end would yield an independent, unshared input for each multiplexer, so that there would now be two 4:1 multiplexers sharing 3 inputs. In general, two $N:1$ multiplexers stacked this way will share $(N-1)$ inputs if N is even, and all N inputs if N odd. In the UltraScale network we find examples of both shared 4:1 and 6:1 multiplexers, which we would expect to share all-but-one input, and that is what we see in the data.

Third, some entity names include a "W" or "E" character, which appear to indicate that they map to the East or West side of the interconnect tile respectively. These likely correspond to the slices that appear on the East and West sides of the interconnect block in

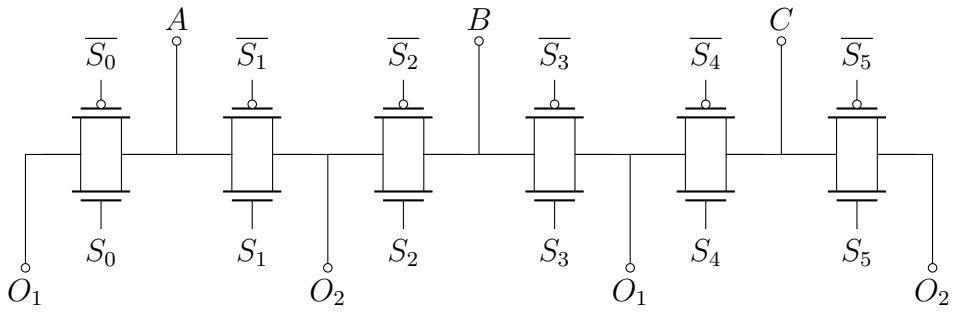


Figure 3.11: Circuit view of two 4:1 transmission-gate multiplexers sharing 3 inputs

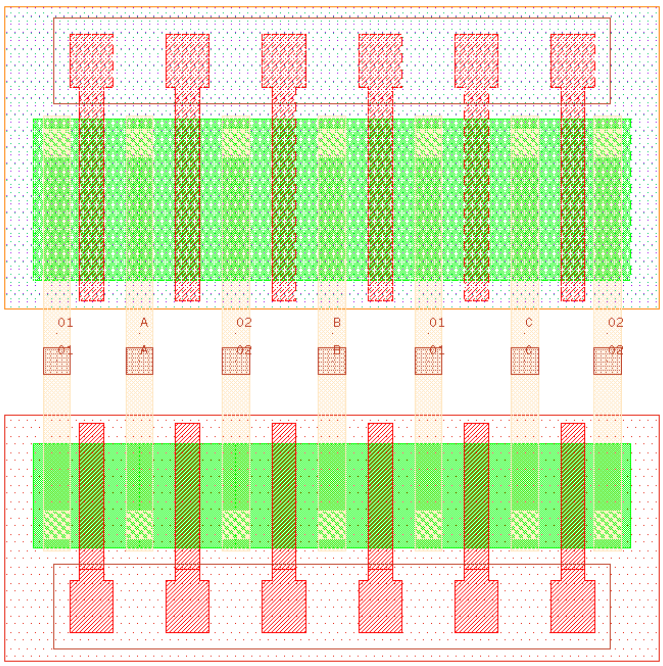


Figure 3.12: Layout view of two 4:1 transmission-gate multiplexers sharing 3 inputs

Figure 3.4. More interestingly, some groups appear to repeat for each side, but others appear to be shared between them. The first shared group is the interconnect crossbar which, as we will discuss in the Section 3.6, is more or less independent of each slice. The second shared group is the second stage of the OIB which, as we will see in Section 3.5, contains many cross-tile connections, and would make sense to place somewhat centrally between the East and West slices.

3.3 Interconnect Wiring

In the UltraScale architecture, inter-tile connectivity is afforded by direct, uni-directional tile-to-tile wires. Lemieux et. al. have shown [60] that uni-directional wiring is a good idea, and here is commercial validation. Each wire has a single driver in the source tile but drives mostly nearby sinks in the receiving tile. The limited number of exceptions to this act to distribute a wire across receivers in vertically adjacent tiles.

The totality of inter-tile interconnect wiring is summarised in Table 3.2.

3.4 Input Interconnect Block

The Input Interconnect Block (IIB) is responsible for the distribution of incoming signals to the inputs of the BLEs. The majority of these are incoming interconnect wiring, but there are also numerous feedbacks paths from the outputs of the BLEs in the same slice.

The first stage lands the inputs from all incoming interconnect wiring and feedback paths from the output stage: it is the first "fly" of what can be loosely described as a 6-ary 2-fly Butterfly-ish network formed with the second stage. The IIB structure is mirrored on the east and west sides of the interconnect block. Each side is responsible for its respective slice within the tile, so we will limit our analysis to the east slice in what follows.

Stage 1 contains 64× 6:1 shared multiplexers (32× physical multiplexer structures) named INT_NODE_IMUX_x_INT_OUTy (see Table 3.1). Notably, the space of Stage 1 ordinals is shared between the eastern and western slices, which have $x \in [0, 31)$ and $x \in [32, 63)$ respectively. This is in contrast to the numbering schemes in other regions of the network that explicitly distinguish the eastern and western slices (with an "E" or a "W"). The majority of nodes in Stage 2, on the other hand, are dedicated 6:1 multiplexers (with no shared inputs) named IMUX_{E,W}x. Since we consider only the east side here, ours will be named IMUX_Ex. There are 64 nodes in this stage: one for each of the 8 input pins on all 8 of the BLEs.

Stage 1 also contains eight "non-local" bounce nodes named INODE_{E,W}_x_FTy per slice that connect to IIB Stage 2 multiplexers both in the current tile and in the neighbouring tiles to the north and south. They are numbered $x \in \{1, 3, 5, 9, 54, 58, 60, 62\}$, which is at-once strange because here, the ordinals are explicitly distinguished between the east and west slices, whereas we just saw that for regular Stage 1 nodes they are not. One explanation for this is that these ordinals instead number *logical* multiplexers within Stage 1 of the IIB. In fact, the set of regular Stage 1 nodes is actually missing four logical multiplexers 0_B, 11_B, 22_B and 31_B, as we will see Figure 3.14. When we check for input sharing, we find that some of the non-local bounce nodes are in fact the missing Stage 1 multiplexers. Pairs such as INODE_E_1_FT1 and INT_NODE_IMUX_0_OUT0(0,0) (which we call 0_A) share 5 of their 6 inputs (Fig. 3.13), indicating that former is using the logical multiplexer that would have been labelled 0_B.

Table 3.2: The interconnect wiring in an UltraScale tile

Count	Length (# tiles)	Direction	Source side	Source (dX, dY)	Prefix
8	12	West		6, 0	WW12
8	4	West	West	2, 0	WW4_W
8	4	West	East	2, 0	WW4_E
8	2	West	West	1, 0 (1)	WW2_W
8	2	West	East	1, 0 (1)	WW2_E
8	1	West	West	1, 0	WW1_W
1	1	West	East	0, 0	WW1_E_7_FT [1]
8	12	South		0, 12	SS12
8	4	South	West	0, 4 (5)	SS4_W
8	4	South	East	0, 4 (5)	SS4_E
8	2	South	West	0, 2	SS2_W
8	2	South	East	0, 2	SS2_E
8	1	South	West	0, 1	SS1_W
8	1	South	East	0, 1	SS1_E
8	12	North		0, -12	NN12
8	4	North	West	0, -4 (-5)	NN4_W
8	4	North	East	0, -4 (-5)	NN4_E
8	2	North	West	0, -2	NN2_W
8	2	North	East	0, -2	NN2_E
8	1	North	West	0, -1	NN1_W
8	1	North	East	0, -1	NN1_E
8	12	East		-6, 0	EE12
1	12	East		-6, 1	EE12
1	12	East		-6, -1	EE12
8	4	East	West	-2, 0	EE4_W
8	4	East	East	-2, 0	EE4_E
8	2	East	West	-1, 0	EE2_W
8	2	East	East	-1, 0	EE2_E
8	1	East	East	-1, 0	EE1_E

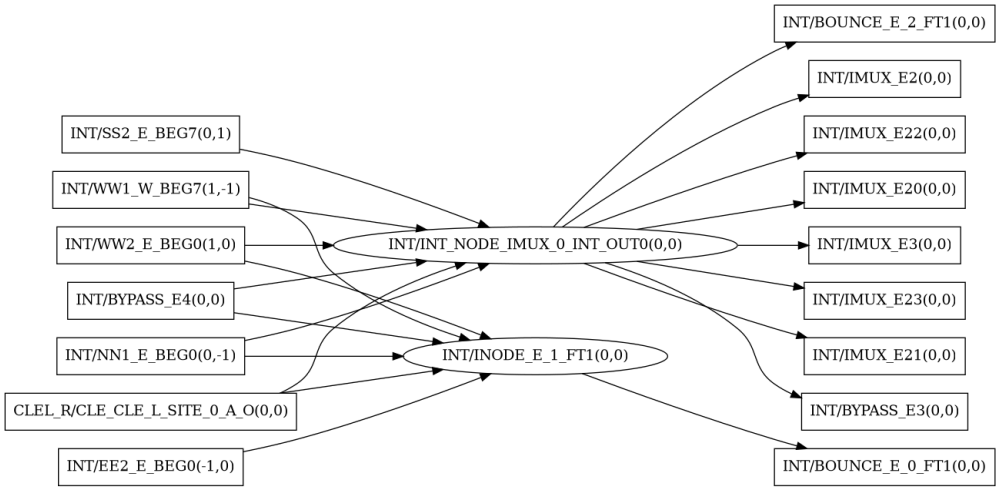


Figure 3.13: A non-local bounce node using the other logical side of a regular Stage 1 multiplexer

Table 3.3: Correspondence of non-local IIB Stage 1 nodes to regular IIB Stage 1 nodes

Non-local bounce node	... is the other side of node
INODE_E_1_FT1(0,0)	INT_NODE_IMUX_0_INT_OUT0(0,0)
INODE_E_3_FT1(0,0)	INT_NODE_IMUX_11_INT_OUT0(0,0)
INODE_E_5_FT1(0,0)	INT_NODE_IMUX_22_INT_OUT0(0,0)
INODE_E_9_FT1(0,0)	INT_NODE_IMUX_31_INT_OUT0(0,0)
INODE_E_58_FT0(0,0)	INODE_E_60_FT0(0,0)

The full correspondence betweeen non-local bounce nodes and logical Stage 1 multiplexers is listed in Table 3.3. We note that two of the non-local bounce nodes, 58 and 60, occupy their own separate shared 6:1 multiplexer, but that this is not true of 54 and 62. It is unknown whether {54, 62} are standalone 6:1 muxes or share inputs with nodes missing from the data, such as for control signalling. Assuming the former, non-local bounce nodes increase the total count of physical Stage 1 multiplexers per slice to $32 + 3 = 37$.

3.4.1 Fan-in to Stage 1

The vast majority of connections to Stage 1 multiplexers in the IIB come from interconnect wires of lengths 1 and 2 landing on the same side of the tile. This is shown in the adjacency matrix of Figure 3.14. Because the topology on the western side of the tile is a mirror of the topology of the eastern side, western nodes are omitted from the figure. Additional connections are made from "bounce" multiplexers in the neighbouring tiles and "bypass" multiplexers in the current tile, but these will be discussed in subsequent sections.

The input patterns to the regular Stage 1 multiplexers and the feedback nodes are distinct from one another. In the case of the former, incoming interconnect wires from each direction are regularly distributed across nearby nodes, stepping through wires in each incoming bundle and strafing across output multiplexers with regularity. Fan-out is limited to 4. Most of the connections to Stage 1 multiplexers are to both sides of a shared physical multiplexer, suggesting that one of the shared input pins is used.

A notable exception to this pattern is that *one* wire from each interconnect bundle is used to drive some subset of multiplexers within the "special" ordinals $\sigma_j^{(I)} = \{11, 22\}$. Additionally, the span of the pattern's repetition does not seem to neatly divide available range of ordinals, so at the end of the range a final phase is needed to fix up loose connections.

Interestingly, the east-side of the tile is driven by incoming east-faring wires of length 2, but *not* of length 1. It is driven by length-1 wires travelling *westward from the east* instead. This is probably a simple consequence of the physical layout: wires traveling west will encounter the eastern side of a tile before its western side. Doing this minimises the length of the short wires and speed up local connectivity.

3.4.1.1 Algorithm

A key goal in developing the algorithm for each sub-section of the topology was to understand and extract any generating parameters. Notwithstanding the common folly of trying to extrapolate from a single example, these parameters should enable us to build a smaller interconnect with properties similar to UltraScale example, as we will see in Section 4.10.

Bundles of interconnect wires entering Stage 1 exhibit clear repeating patterns among wires coming from the same direction. That is, it seems that bundles of north-faring wires of length 1 and of length 2 are distributed according to one pattern, bundles of east-faring wires of length 1 and length 2 wires and distributed according to another pattern, and so on for south-faring and west-faring wires.

Let $\rho^{(d)}$ denote the pattern for wires input from direction $d \in \{N, E, S, W\}$ (north, east, south, west), and $\rho_k^{(d)}$ denote the kth instance of that pattern. As an example, consider instances of the pattern for north-faring wires, $\rho_k^{(N)} : k \in \{1, 2, 3\}$, which are drawn as boxes on Figure 3.14. Within a pattern (box), there are three phases: in each phase consecutive output muxes are allocated a wire from either the $N^{(1)}$ bundle, the $N^{(2)}$ bundle, or both. Between phases a certain number of output nodes is skipped. After three patterns are stamped out, no more will fit, and a cleanup step is performed to connect the remaining.

Table 3.4: Synopsis of algorithmic steps when connecting interconnect wiring to IIB Stage 1 multiplexers

(a) North wires, $\rho^{(N)}$

	Phases				
	j	$j \leftarrow j+1$	$j \leftarrow j+1$	$j \leftarrow j+2$	$j \leftarrow j+1$
$N^{(1)}$	$i \rightarrow j_A, j_B$	$i \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$		
$N^{(2)}$	$i \rightarrow j_A, j_B$	$i \rightarrow j_A, j_B$		$(i+1) \rightarrow j_B, (j+1)_A$	$(i+1) \rightarrow j_B, (j+1)_A$

(b) East wires, $\rho^{(E)}$

	Phases					
	j	$j \leftarrow j+1$	$j \leftarrow j+2$	$j \leftarrow j+1$	$j \leftarrow j+2$	$j \leftarrow j+1$
$E^{(1)}$	$i \rightarrow j_A, j_B$	$i \rightarrow j_A, j_B$			$(i+1) \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$
$E^{(2)}$			$(i+1) \rightarrow j_B, (j+1)_A$	$(i+1) \rightarrow j_B, (j+1)_A$	$(i+2) \rightarrow j_A, j_B$	$(i+2) \rightarrow j_A, j_B$

(c) South wires, $\rho^{(S)}$

	Phases					
	$j \leftarrow j+2$	$j \leftarrow j+1$	$j \leftarrow j+2$	$j \leftarrow j+1$	$j \leftarrow j+1$	$j \leftarrow j+1$
$S^{(1)}$			$(i+1) \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$	$(i+2) \rightarrow j_A, j_B$	$(i+2) \rightarrow j_A, j_B$
$S^{(2)}$	$i \rightarrow j_B, (j+1)_A$	$i \rightarrow j_B, (j+1)_A$	$(i+1) \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$		

(d) West wires, $\rho^{(W)}$

	Phases					
	j	$j \leftarrow j+1$	$j \leftarrow j+2$	$j \leftarrow j+1$	$j \leftarrow j+1$	$j \leftarrow j+1$
$W^{(1)}$			$i \rightarrow j_A, j_B$	$i \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$
$W^{(2)}$	$i \rightarrow j_B, (j+1)_A$	$i \rightarrow j_B, (j+1)_A$	$(i+1) \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$		

Table 3.5: Final clean-up phase for north wires, $N^{(1)}$, $N^{(2)}$, after completing $\rho^{(N)}$

	j	$j \leftarrow 2$	$j \leftarrow 1$
$N^{(1)}$	$i \rightarrow j_B, (j+1)_B$	$(i+1) \rightarrow j_B$	$(i+1) \rightarrow (j+1)_A$
$N^{(2)}$	$i \rightarrow j_B, (j+1)_B$	$(i+1) \rightarrow (j+1)_A, (j+1)_B$	$(i+1) \rightarrow j_A$

Table 3.6: Pre- and post-pattern parameters for $\rho^{(d)} : d \in \{N, E, S, W\}$

	i_0	j_0	Δi	Δj
North, $\rho^{(N)}$	1	1	2	2
East, $\rho^{(E)}$	1	1	2	1
South, $\rho^{(S)}$	1	1	2	1
West, $\rho^{(W)}$	1	0	2	1

If we split each phase so that at most two muxes are connected per step, this general structure can describe the pattern for all four wire directions, thus summarised in Tables 3.4a - 3.4d. In each pattern, i is the ordinal of the driving (incoming) interconnect wire being connected, and j is the ordinal of the physical Stage 1 mux being connected to. We use A and B subscripts to denote the first or second logical mux at that ordinal, respectively. The shorthand $i \rightarrow j_A, j_B$ denote creating a connection from wire i to the A and B sides of physical Stage 1 multiplexer j, where each side is a logical multiplexer. Increment operations on j, such as $j \leftarrow j + 1$, implicitly skip over the special ordinals $\sigma_j^{(I)}$. *Any* operations on i and j are cyclic over their respective range of ordinals: incrementing i wraps at 8, and incrementing j wraps at 32.

The final cleanup step differs slightly across each pattern, but the objective is always to make sure that every incoming wire in each bundle drives at least one Stage 1 mux. As an example, the cleanup phase for north wires $N^{(1)}$, $N^{(2)}$, after $\rho(N)$, is shown in Table 3.5.

Importantly, the ordinals j and i are preserved between invocations of these patterns, which are sequentially executed acrosss the full range of Stage 1 multiplexers for each wire direction, $j \in [0, 31]$. As well as the initial step of setting the starting i_0 and j_0 for i and j (respectively), we have a post-instantiation step to increment i and j by Δi and Δj (respectively) for the next round. Table 3.6 lists the values of these parameters for each pattern ρ.

We will now consider the special connections, which we have seen connect to Stage 1 mux ordinals in $\sigma_j^{(I)} = \{11, 22\}$. In all cases we observe that the connections are made from the first ($i = 0$) wire within an incoming interconnect bundle, and that they connect to the A side of the shared multiplexer. There are *no* special connections from north-faring wires. The remaining connections are summarised in Table 3.7.

The full code, in `interconnect_generator/interconnect_to_iib_1.py`, shows the final clean up phases for the other wire directions, and also the allocation of special nodes.

Table 3.7: Special connections from incoming interconnect wiring to IIB Stage 1 multiplexers

Wire direction, size	Source side	Wire index, i	Destination side	Stage 1 mux, j
$E^{(2)}$	East	0	East	$11_A, 22_A$
$E^{(2)}$	East	0	West	$11_A, 22_A$
$W^{(1)}$	West	0	East	$11_A, 22_A$
$S^{(1)}$	East	0	East	$11_A, 22_A$
$S^{(2)}$	West	0	West	$11_A, 22_A$

3.4.2 Stage 1 to Stage 2

The core connectivity of the 2-fly IIB is captured in the fan-out from Stage 1 to Stage 2. To understand it, we first assembled the adjacency matrix for the network, shown in Figure 3.15. The default natural sorting of node names immediately provides several hints about patterns within the fan-out:

1. On average, later ordinals in the first stage drive later ordinals in the second stage, but this pattern repeats across four distinct groups: the bypass nodes, bounce nodes, and each of two partitions of the main BLE nodes.

2. Alternating rows seem to share connectivity, suggesting that the Stage 1 drivers are grouped.

3. The first rows seem to continue the pattern of the last rows, suggests some "wrap-around".

To the reader appraised of Linear Algebra, and perhaps even of Spectral Graph Theory, the next most obvious thing to do with our adjacency matrix is to attempt some sort of row-reduction. In practice, this required sorting by each successive column to try and get the matrix into upper triangular form, which immediately elucidated the groupings of Stage 1 nodes. We sorted these groups in order of the ordinals to which they connect, and then swapped rows until the difference between successive rows was minimal.

The reordered adjacency matrix is shown in Figure 3.16. Here we can clearly see the pattern of linearly progressing connectivity Stage 1 to Stage 2, and consistent groupings of the Stage 1 nodes into (almost always) triplets that share connectivity. The four Stage 2 groupings are:

A: 4× Bounce muxes (6.25%), named BYPASS_Ex for $x \in \{0, 2, 13, 15\}$

B: 12× Bypass muxes (12.5%), named BOUNCE_Ex for $x \in (0, 15), x \notin \{0, 2, 13, 15\}$

C: 16× Regular muxes (25%), named IMUX_Ex for $x \in [0, 15]$

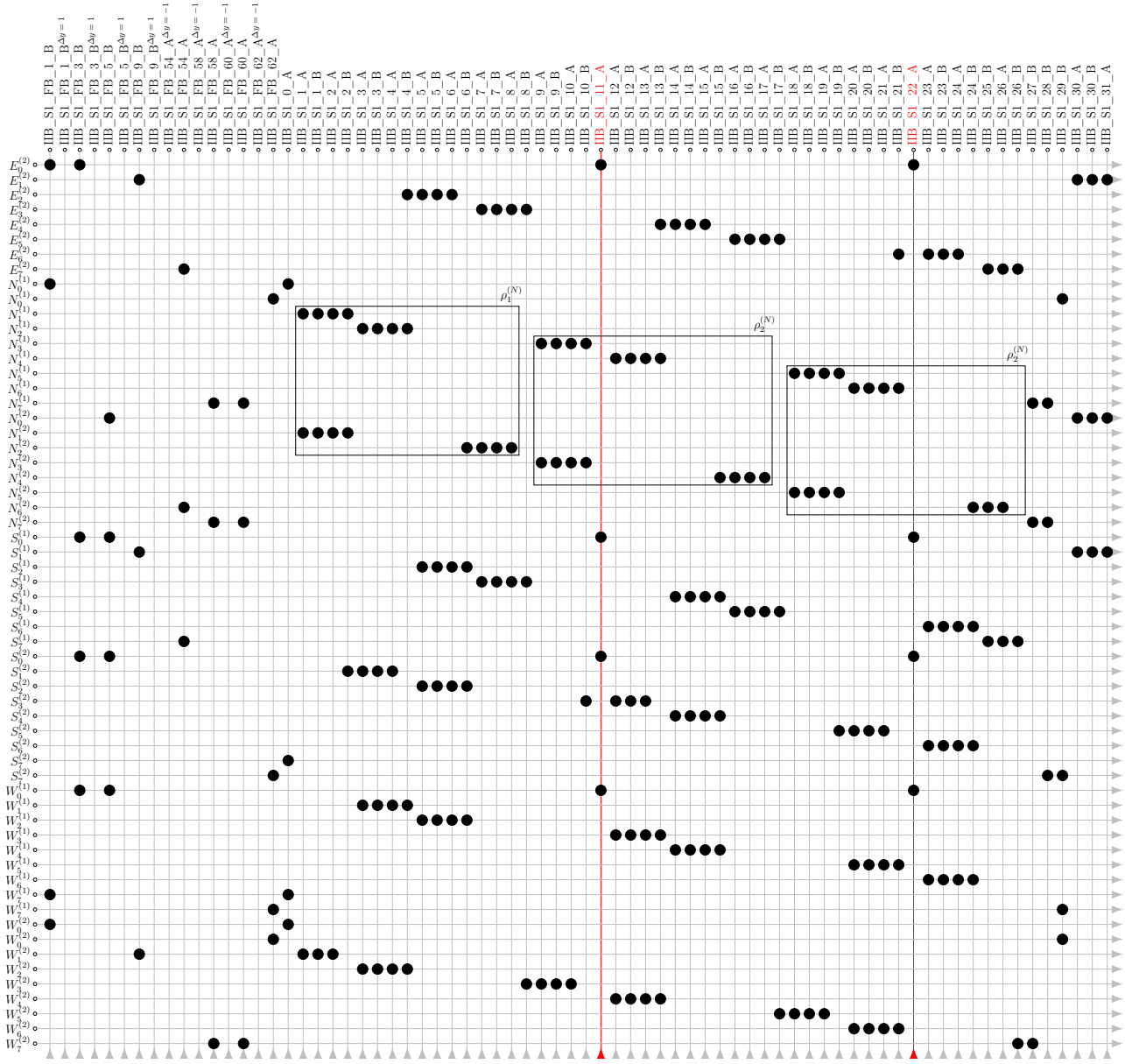


Figure 3.14: Interconnect to IIB S1

D: 32× Regular muxes (50%), named IMUX_Ex for $x \in [16, 47]$

Group A is driven by low- and high-ordinal Stage 1 nodes, and in particular by the Stage 1 feedback nodes in this and vertically neighbouring tiles. The fact that low-ordinal bounce nodes 0, 2 are driven by neighbouring nodes from the south ($\Delta y = -1$) and high-ordinal nodes 13, 15 are driven by neighbouring nodes from the north ($\Delta y = 1$) reaffirms the suspicion from Section 3.2.1 that nodes are numbered accordingly with their physical

location across the tile. Additionally, since nodes from the south are numbered at the high end of the range (i.e. 54, 58, 60, 62) and connect to low numbers (0, 2), and the opposite is true for northern feedback nodes, we can conclude that lower-number ordinals appear near the south and higher number appear near the north. This would serve to minimise wire length and/or path congestion.

Group B is the first to demonstrate repeating connection sub-clusters across the remaining groups. Consider the boxed regions drawn on Figure 3.16: these are regions of high-connectivity that are isolated from the rest of the graph, and which repeat across the input and output node range. We call these "cliques" for short, even though they are not fully connected (in the graph theoretic sense). In each "clique", every Stage 2 node is driven by exactly two groups of Stage 1 nodes. There is no connection between the Stage 1 and 2 nodes within the group but outside of the clique.

Groups C and D largely mirror the pattern of group B, but with additional connections at the lowest- and highest-ordinal nodes. There is also some deliberate shuffling: now a single driver from the last output node is disconnected, and in its place a driver from the next clique is connected. We suspect that this is to add path diversity, or entropy, to the routing graph.

Group D also doubles the number of outputs over group C, and so each clique has twice as many sinks. Stage 1 drivers now simply drive a neighbouring pair of Stage 2 nodes instead of a single one. Under our hypothesis of physically-aware ordinal numbering these nodes are close together, so both the wire cost of driving the pair and the congestion introduced are limited.

With notable exceptions, the three drivers in each Stage 1 grouping always form a sequence of drivers 2 apart. This means that three wires from nearby drivers go to the same places and can be routed together. We also observe that the groupings are the same between outputs of shared muxes: the same physical multiplexers appear in one group using their first output and in another group using their second. Table 3.8a shows how the *regular* (non-feedback) physical IIB Stage 1 multiplexers are successively allocated to groups.

There are five special groups, two of which appear in the table: a) group 0, which consists of drivers at the start, approximately middle and end of the range {0, 22, 31}; and b) group S, which consists of {11, 28}. If we imagine that these are allocated first and the regularly-observed patterns are applied to the remaining nodes, we recover the groupings observed in the data. (The reader will recognise the ordinals $\{11, 22\}$ from the fan-in patterns to Stage 1, where they they were also exceptional nodes $\sigma_j^{(I)}$ - more on this later.) Three groups of feedback nodes do not appear — they connect to Stage 2 nodes in each group physically nearest their source.

3.4.2.1 Algorithm

Before connectivity can be assigned between Stage 1 and Stage 2, the Stage 1 multiplexers must be put together in groups, and the groups must be ordered for allocation. This

Algorithm 1 Assemble IIB Stage 1 groups

$L \leftarrow$ number of Stage 2 multiplexer inputs (6)
$S \leftarrow \lfloor L/2 \rfloor$ ▷ the size of each Stage 1 group
procedure SPECIALGROUPS(X)
  $C \leftarrow \big[X[i \cdot \lceil |X|/S \rceil] : i \in [0, S)\big] +\!\!+ [X[-4],\, X[-1]]$
  $\alpha \leftarrow [\bot]^{|C|}$
  for $i \in [0, |C|)$ s.t. $\alpha[i] = \bot$ **do**
    $I \leftarrow [i,\, i{+}2,\, \ldots]$ up to length S
    if $|I| = S$ **then**
      Emit $\langle C[j] : j \in I \rangle$; $\alpha[j] \leftarrow \top\ \forall j \in I$
    end if
  end for
  return groups, $\{C[i] : \alpha[i] = \bot\}$
end procedure
procedure LINEARGROUPS(X)
  Partition prefix of X into non-overlapping S-tuples
  return groups, $X[S\lfloor |X|/S \rfloor :]$
end procedure

$(G_\sigma,\, U_\sigma) \leftarrow$ SPECIALGROUPS(X_r) ▷ X_r: regular mux ordinals
$U \leftarrow X_r \setminus (G_\sigma \cup U_\sigma)$ ▷ unallocated ordinals
for each $g \in$ LINEARGROUPS(X_r) **do** ▷ regular groups from U in steps of 2 mod $|U|$,
  Walk U in steps of 2; skip $\min(U)$ when $|g| = S - 1$ ▷ avoiding $\min(U)$ in the last slot
  Emit g; remove its elements from U
end for
$P_0[i] \leftarrow$ primary S-tuple for group i
$P_1[i] \leftarrow$ input-sharer S-tuple ▷ P_1 omitted for last group

$\Gamma \leftarrow$ LINEARGROUPS(X_ℓ) $+\!\!+$ LINEARGROUPS(X_n) ▷ local, north feeders first
$\kappa[i] \leftarrow 0\ \forall i$; $T \leftarrow 3$; $i \leftarrow 0$ ▷ spacing clock, trigger threshold
while $i < |P_0|$ **do**
  if $(|\Gamma| - 4) \bmod 16 = 0$ and $G_\sigma \neq \emptyset$ **then** ▷ insert special group at fixed intervals
    Γ += pop(G_σ)
  else if $\max(\kappa) \geq T$ **then** ▷ force paired *out1* group when clock saturates
    $j \leftarrow \arg\max(\kappa)$; Γ += $P_1[j]$; $\kappa[j] \leftarrow 0$
  else
    Γ += $P_0[i]$; $\kappa[i] \leftarrow 0$; i += 1 ▷ advance through primary groups
  end if
  $\kappa[j]$ += $1\ \forall j < i$
end while
Append remaining P_1 in index order; Γ += LINEARGROUPS(X_s) ▷ south feeders last
return Γ

Algorithm 2 Preparation and distribution of IIB S1 muxes over IIB S2 muxes

— **Phase 1: Mux distribution** —
$r_{g,k} \leftarrow$ fraction of IIB S2 muxes in group g cliques of size k
$\mathbf{R} \leftarrow [r_{g,k}]_{4\times 3}$
$\mathbf{D} \leftarrow \lfloor \mathrm{diag}(\mathbf{n}^{(2)}) \cdot \mathbf{R} \rfloor, \quad \mathbf{n}_g^{(2)} = |G_g^{(2)}|$ ▷ $\mathbf{D}_{g,k}$: S2 muxes in group g for size-k cliques

— **Phase 2: Clique formation** —
for $g \in \{A, B, C, D\}$ **do**
  $m \leftarrow 2$ if $g = D$, else 1 ▷ mux multiplicity per clique slot
  $n_k \leftarrow \mathbf{D}_{g,k}/(k \cdot m)$ for $k \in \{3, 2, 1\}$ ▷ number of size-k cliques
  $Q \leftarrow \mathrm{deque}(G_g^{(2)})$; $\mathcal{C}_g \leftarrow [\,]$ ▷ each clique entry is an m-tuple of muxes
  if $n_2 > 0$ **then**
    Pop $\lceil n_2/2 \rceil$ size-2 cliques from front of Q into $\mathcal{C}_g$
  else if $n_1 > 0$ **then**
    Pop $\lceil n_1/2 \rceil$ size-1 cliques from front of Q into $\mathcal{C}_g$
  end if
  Pop n_3 size-3, then $\lfloor n_2/2 \rfloor$ size-2 cliques from Q into $\mathcal{C}_g$
  Pop n_1 size-1 cliques into $\mathcal{C}_g$ ▷ all n_1 if $n_2 > 0$; else only $\lfloor n_1/2 \rfloor$
end for

— **Phase 3: Input-group assignment** —
for $g \in \{A, B, C, D\}$, clique $c \in \mathcal{C}_g$ **do**
  $k \leftarrow$ the size of clique c
  $F \leftarrow k$ if IsShort(g, k), else $k + 1$ ▷ short cliques in C/D get 1 fewer input group
  $I_{0..F-1} \leftarrow$ next F S1 groups (cyclic over $\mathcal{G}^{(1)}$)
  for $j \leftarrow k - 1$ **downto** 0 **do** ▷ sliding 2-group window over clique members
    $\kappa \leftarrow j + (F - k)$
    for all mux u in tuple c_j **do**
      $E \mathrel{+}= I_\kappa \times \{u\}$
      if $\kappa > 0$: $E \mathrel{+}= I_{\kappa-1} \times \{u\}$
    end for
  end for
end for

Table 3.8: IIB Stage 1 multiplexer allocations to groups of like drivers

(a) As observed in UltraScale

IIB S1 Mux #	Group allocation										
0	0										
1		1									
2			2								
3		1									
4			2								
5				3							
6			2								
7				3							
8					4						
9				3							
10					4						
11	S										
12						5					
13					4						
14						5					
15							6				
16						5					
17							6				
18								7			
19							6				
20								7			
21									8		
22	0										
23								7			
24									8		
25										9	
26									8		
27										9	
28	S										
29										9	
30		1									
31	0										

(b) A naive algorithm

IIB S1 Mux #	Group Allocation										
0	0										
1		1									
2			2								
3		1									
4			2								
5				3							
6			2								
7				3							
8					4						
9				3							
10					4						
11	S0										
12					4						
13						5					
14							6				
15						5					
16							6				
17						5					
18							6				
19								7			
20									8		
21								7			
22	0										
23								7			
24									8		
25										9	
26									8		
27										9	
28	S1										
29										9	
30		1									
31	0										

step is done at set up by the `assemble_iib_s1_groups` method in `interconnect_generator/architecture.py`. A sketch is provided in Algorithm 1.

A natural starting place for the IIB Stage 1 group allocation is to first remove special

— **Phase 4: C–D cross-shuffle perturbation** —
for $g \in \{C, D\}$, clique index j **do**
  $\alpha_{g,j} \leftarrow$ first mux of $I_{F-2}^{(g,j)}$; $\beta_{g,j} \leftarrow$ last mux of $c_{F-1}^{(g,j)}$
  $E \mathrel{-}= \{(\alpha_{g,j},\, \beta_{g,j})\}$ ▷ sever penultimate-group → tail-mux edge
end for
for $j = 0$ **to** $|\mathcal{C}_D| - 1$ **do**
  if $\alpha_{C,j+1}$ exists $\wedge\, (\alpha_{C,j+1}, \beta_{D,j}) \notin E$ **then**
    $E \mathrel{+}= \{(\alpha_{C,j+1},\, \beta_{D,j})\}$
  end if
  $t \leftarrow j$ if $(\alpha_{D,j}, \beta_{C,j}) \notin E$, else $j - 1$
  if $\beta_{C,t}$ exists **then**
    $E \mathrel{+}= \{(\alpha_{D,j},\, \beta_{C,t})\}$
  end if
end for

— **Phase 5: Realize edges & rebalance** —
Connect all $(u, v) \in E$ in the routing graph
repeat ▷ Fan-out equalization; local feedbacks $\mathcal{L}$ are exempt
  $n^* \leftarrow \arg\max_{n \notin \mathcal{L}} |\mathrm{out}(n)|$; $s^* \leftarrow \arg\min_{n \notin \mathcal{L}} |\mathrm{out}(n)|$
  if $|\mathrm{out}(n^*)| \le \Delta_{\max}$ **or** $|\mathrm{out}(s^*)| > \Delta_{\max}$ **then break**
  end if
  Pick any $v \in \mathrm{out}(n^*)$; disconnect $n^* \to v$; connect $s^* \to v$
until converged
for each v with $|\mathrm{in}(v)| > \Gamma_{\max}$ **do** ▷ Fan-in cap
  Disconnect $|\mathrm{in}(v)| - \Gamma_{\max}$ arbitrary inputs from v
end for

nodes and then allocate sequences of $S_I = 3$ multiplexers, with ordinals 2 apart, to successive groups. Each allocation step starts at the first unallocated Stage 1 multiplexer, and any previously-unallocated muxes are skipped. Now, we arrive at the naive allocation of Table 3.8b, which differs from the actual allocation we observe in key ways. In the UltraScale, the ordinal range covered by any Stage 1 group:

- overlaps with at least one other group, adding path diversity; and
- is limited to 5, not 6, reducing the total wire length (under the hypothesis of physically numbered devices).

To ensure these additional invariants, we modify the algorithm as follows: when picking the last member of a group, if picking the default element would leave no unallocated ordinals before the next one, pick the one after it instead. As an example, consider that when assigning

the last member of group 4, choosing mux #12 would leave no unallocated IIB Stage 1 muxes in the range $[0, 12]$. So choose 13 instead. This ensures overlap: when group 5 is allocated, it will pick up #12 instead.

The "special" Stage 1 multiplexers are the feedback nodes and, from the list of regular nodes:

1. $S_I + 1$ equally-spaced entries across the entire range, at indices

$$\left\{0, \left\lfloor \frac{N_I}{S_I} \right\rfloor, 2 \left\lfloor \frac{N_I}{S_I} \right\rfloor, \ldots, (N_I) \left\lfloor \frac{N_I}{S_I} \right\rfloor - 1\right\}$$

2. the 3rd to last entry.

The last step in assembling Stage 1 groups is to place them in the right order for allocation in the next phase, cognizant of the fact that each group of physical multiplexers actually represents two logical ones (for each of the two outputs). Denoting the groups by their ordinal, we suffix "B" to distinguish the second logical multiplexer grouping from the first. Then, reading from the rows in Figure 3.16, the order they appear in is:

$$1, 2, 0, \{3, 1B, 4, 2B\}, \{3B, 5, 6, 4\}, \{7, 5B, 8, 6B\}, 7, 9, 8B$$

A notable property of this pattern is that the second logical output for a particular group appears about 4 or 5 positions after the first, so it is always in a different clique. (The parentheses group together Stage 1 logical groups that participate in the same clique during allocation to Stage 2 drivers.) We use a "clock" mechanism to recreate this order: for each physical Stage 1 grouping, a "clock" tracks how many steps have occurred since allocation from that group. As a pointer scans upward through the group ordinals, each step advances the clock of every ordinal less than the current pointer. (That this, the clock does not start until the ordinal's first turn, and only clocks that have been started are ticked over.) On any iteration, if any ordinal's clock has exceeded the trigger value, it is inserted into the output order immediately, and its clock is reset. The first time an ordinal is allocated we use its first logical output; the second time, we flip to its second logical output.

The full algorithm for generating IIB Stage 1 to Stage 2 connectivity is provided in `interconnect_generator/iib_1_to_2.py` within the BFG source code. A sketch is provided in Algorithm 2. There are broadly five phases to the algorithm:

1. Determine, from configured parameters, the size and composition of each clique within the network. We must determine the number of IIB Stage 2 multiplexers $\mathbf{D}_{g,k}$ participating in group g and clique size k, for $g \in \{A, B, C, D\}$ and $k \in \{3, 2, 1\}$. (The size of a clique k is the number of unique fan-in patterns among its output nodes, the multiplicity m is how many times each of those patterns is repeated.) At this point the feedback multiplexers in Stage 1 are ignored. $\mathbf{D}$ is a simple product of the fixed

ratios of node membership we observe, $\mathbf{R}$, and the diagonal matrix formed from the observed number of nodes in each group $\mathbf{n}^{(2)}$:

$$\begin{aligned}\mathbf{D} &= \operatorname{diag}\left(\mathbf{n}^{(2)}\right)\mathbf{R} \\ &= \begin{bmatrix}4 & & & \\ & 12 & & \\ & & 16 & \\ & & & 32\end{bmatrix}\begin{bmatrix}0 & 0 & 1 \\ 3/4 & 0 & 3/12 \\ 9/16 & 6/16 & 1/16 \\ 24/32 & 8/32 & 0\end{bmatrix} \\ &= \begin{bmatrix}0 & 0 & 4 \\ 9 & 0 & 3 \\ 9 & 6 & 1 \\ 24 & 8 & 0\end{bmatrix}\end{aligned}$$

2. Allocate all Stage 2 multiplexers to cliques based on D from phase (1). At this point the *multiplicity* factor, m, is used to determine how many times each pattern is repeated within a clique. We observed that for group D the multiplicity $m = 2$, but otherwise $m = 1$. Cliques of size $k = 3$ are allocated to the middle of the range, with each half of the cliques of size $k = 2$ and $k = 1$ sandwiched on either side.

3. Allocate incoming Stage 1 groups to each clique $c \in \mathcal{C}_g$, where $\mathcal{C}_g$ is the set of all cliques defined for group g. Given a clique of size k, the default pattern requires $F = k + 1$ input Stage 1 groups.

   To see why $F = k + 1$, we illuminate two previously-hidden parameters. Let L be the size of the output Stage 2 muxes. Let S be the number of Stage 1 muxes in each group being allocated to those Stage 2 muxes. Each node in an input group is allocated to unique pins on an outgoing multiplexer so $\lceil L/S \rceil$-many groups are required to connect all L pins. In the UltraScale, $L = 6$ and $S = L/2 = 3$, so we always require $\lceil L/S \rceil = 2$ incoming groups to saturate an outgoing multiplexer's inputs. The default pattern of connectivity within a clique of size k requires that $(k-1)$ output groups share an input group. There are $N_O = k \cdot L$ pins to drive at the output of the clique (multiplexer inputs), and $N_I = F \cdot S$ unique sources. Removing $S \cdot (k-1)$ duplicate connections to outputs, we require[2]

$$\begin{aligned}N_I &= N_O - S \cdot (k-1) \\ F \cdot \frac{L}{2} &= k \cdot L - \frac{L}{2} \cdot (k-1) \\ F &= 2k - (k-1) \\ F &= k + 1\end{aligned}$$

[2]Another way to see this is to note that the first and last output muxes require an additional input group each but, since these are not shared, this requires 2 more input groups. Therefore, nominally, $F = 2 + k - 1 = k + 1$ incoming groups are required.

There are a couple of exceptions to this, where "short" cliques diverging from the pattern can only accommodate $F = k$ input groups, and they are treated as necessary.

4. Perturb the cliques in groups $g \in \{C, D\}$ to cross-couple their connectivity by cyclically shuffling connections between them. Start with $k = 3$ clique $j = 0$ in group $g = C$, and select both the first Stage 1 mux in the second to last input group $I_{F-2}^{(g,j)}$ and the last output mux in the Stage 2 sinks for that clique. Call these $\alpha_{g,j}$ and $\beta_{g,j}$, respectively. *Disconnect* $\alpha_{g,j}$ from $\beta_{g,j}$, but note both for later. Next, go through all the disconnected sinks $\beta_{g,j}$ and connect the newly disconnected driver from the *previous* group $(g - 1)$ but next *clique* $(j + 1)$, $\alpha_{g-1,j+1}$ instead. Avoid duplicate connections and attempt to reconnect severed connections. If ever a $\beta_{g-1,j}$ is found to already be driven by $\alpha_{g,j}$, connect the driver to $\beta_{g-1,j-1}$ instead.

5. Perform invariant checks, ensuring that the maximum fan-out of Stage 1 is never exceeded (no more than 6), and no Stage 2 multiplexer has too many inputs allocated (also no more than 6). Redistribute to any missing connections opportunistically to make it even. Finally connect unallocated drivers from the irregular Stage 1 feedback nodes.

3.4.3 Stage 2 to BLE inputs

There is a distinct fan-out pattern between each Stage 2 group of multiplexers (A, B, C, D) and the BLE inputs. Overall connectivity is shown in Figure 3.17.

3.4.3.1 Groups A & B

The simplest pattern is that of Group A, the *bounce* multiplexers, which connect to a bypass input on BLEs 0, 2, 5 and 7, i.e. the first, third, third to last and last nodes. However, these nodes also drive Stage 1 multiplexers in neighbouring tiles. An example is shown in Figure 3.18: the first bounce multiplexer in the current slice "(0,0)" drives a single local bypass BLE input. Its corresponding node in the slice above "(0,1)" drives Stage 1 muxes in this tile, so we expect it to do the same for its neighbours.

The pattern among Group B nodes, the *bypass* multiplexers, is similar. The 12 bypass nodes are split into three: the two at the start, the middle eight, and the two at the end. The middle eight are sequentially allocated to a bypass input on each BLE, in order. The first group is allocated a bypass input on BLEs 1 and 3, and the last group is allocated a bypass input on BLEs 4 and 6. Like the bounce nodes, bypass nodes drive Stage 1 nodes as well as their designated BLE input but, unlike bounce nodes, bypass nodes do not drive outside of the current tile.

Between them, groups A and B have 16 nodes for 16 BLE bypass inputs. Central BLEs get both of their bypass inputs from Stage 2 bypass nodes, but peripheral BLEs get one input from a bounce node. The connections are symmetric about the central ordinal. This is

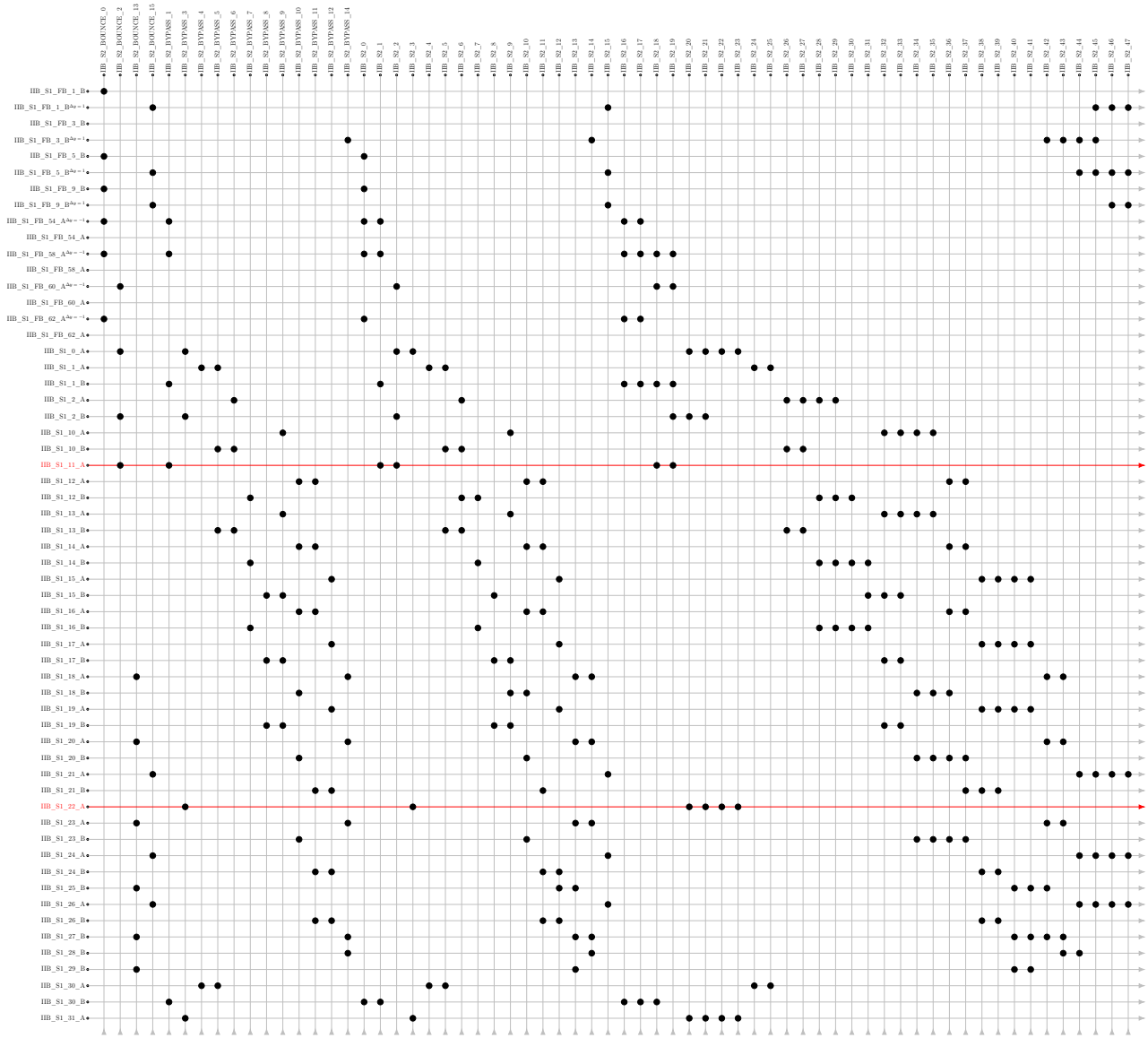


Figure 3.15: IIB S1 to S2

trivial to explain under the assumption of physically-meaningful ordinals: the bounce nodes, which connect to other tiles, are probably near the tile edges in order to reduce wire length and avoid congestion toward the centre of the tile. They connect to the BLEs near the slice edges for the same reasons.

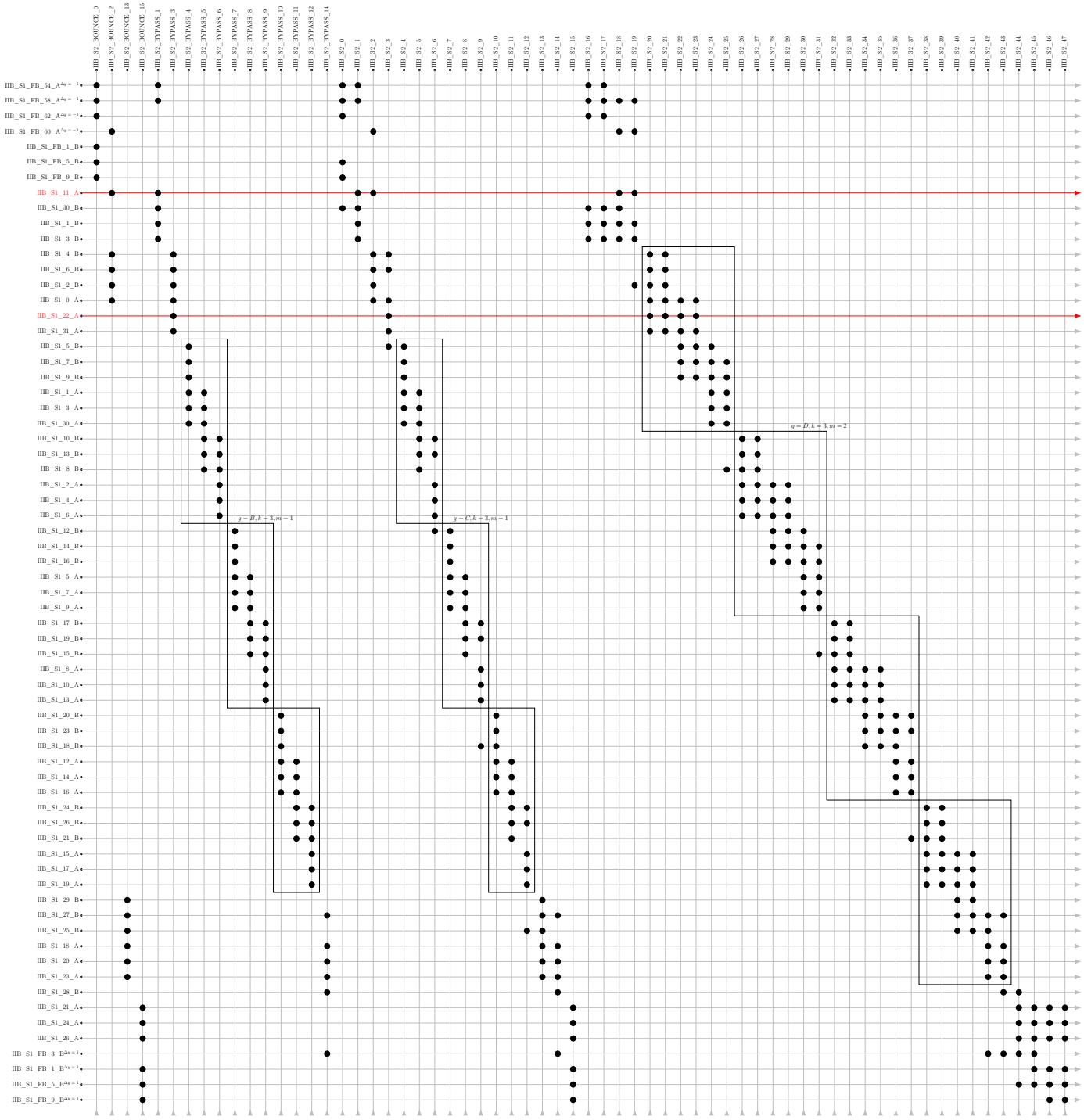


Figure 3.16: IIB S1 to S2 but you reorder the drivers

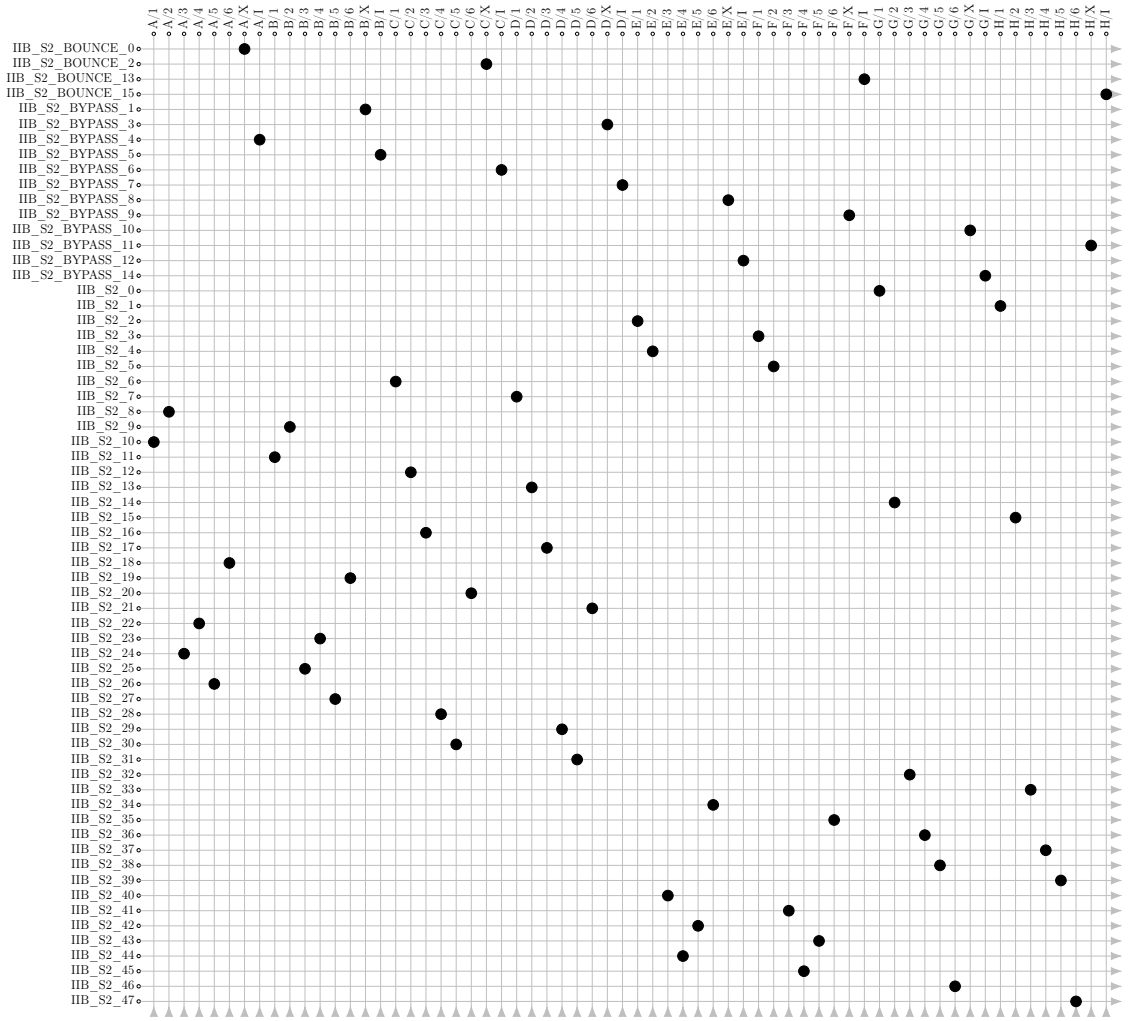


Figure 3.17: IIB S2 driving the BLE inputs

3.4.3.2 Groups C & D

Both groups C and D can independently drive all of the BLEs. The number of pins they have access to is proportional to the group's size: group C has one third of the regular Stage 2 nodes, and group D has two-thirds. Likewise, group C drives two of six BLE inputs (⅓), and group D drives four of six (⅔).

The allocations of group C and D nodes to BLE inputs are reproduced in Tables 3.9 and 3.10, respectively. Each column contains the destination BLE allocated to a single Stage 2 multiplexer, indicated by the ordinal in the first row. Each row contains a new allocation of a single Stage 2 driver to one BLE. The BLEs 0-7 are labelled A-H and coloured to aid the reader. The order of rows pertains to the algorithm we suggest for their allocation, presented in the next section. For now, considering only the final row, we note that the allocations of

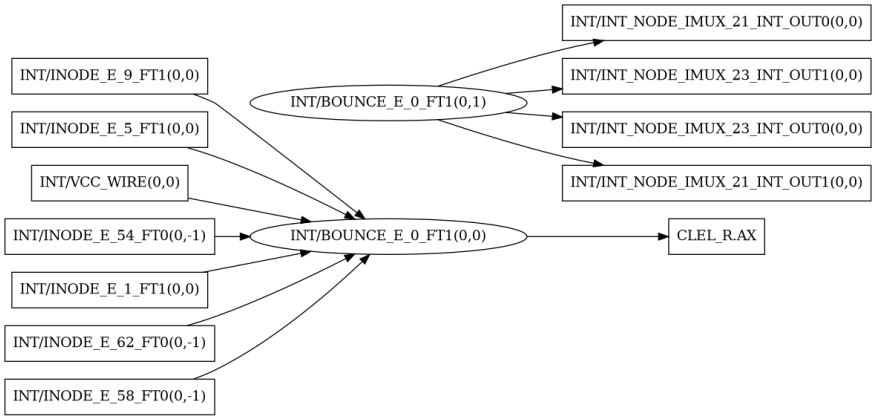


Figure 3.18: Bounce muxes drive a local BLE bypass input and neighbouring Stage 1 muxes

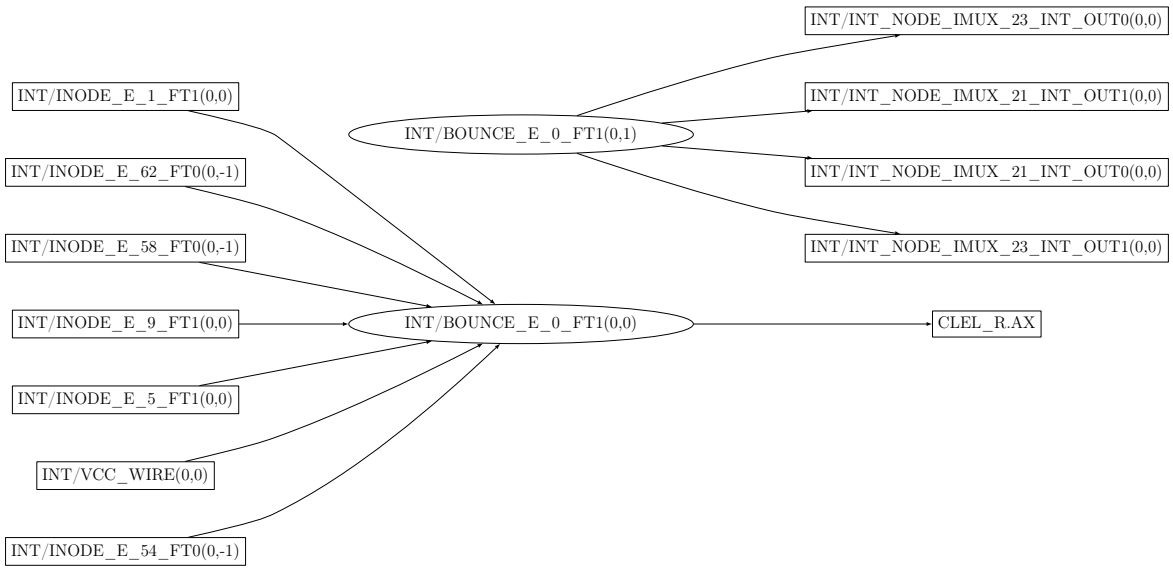


Figure 3.19: TODO: Does this look better than 3.17?

Stage 2 nodes to BLE inputs have two interesting properties:

1. Pairs of multiplexers drive pairs of adjacent BLEs, but adjacent multiplexers never drive the same BLE.

2. Some BLEs get all of their inputs from a short range of (nearby) drivers, while others get them from a more dispersed range.

Why not simply divide the range in half? One possible explanation is that, for functions requiring a limited number of BLE pins, the network can be routed using only the smaller

Table 3.9: IIB Stage 2 to BLE connections, group C

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
← Allocation steps								A								
							B									
										C						
									D							
						A										
					B											
				C												
			D													
												E				
											F					
		G														
	H															
														E		
													F			
																G
															H	
	H	**G**	**D**	**C**	**B**	**A**	**B**	**A**	**D**	**C**	**F**	**E**	**F**	**E**	**H**	**G**

group of Stage 2 nodes. Since we suspect the groups are also placed together physically, using the smaller group should result in lower overall wire delay. In fact, doing so can more generally provide fast, flexible access to the two group C BLE pins, while retaining full connectivity to the other four. Since LUT pins can be reordered arbitrarily, this scheme allows an application's critical path to be mapped to the fast group without compromising overall logic capacity.

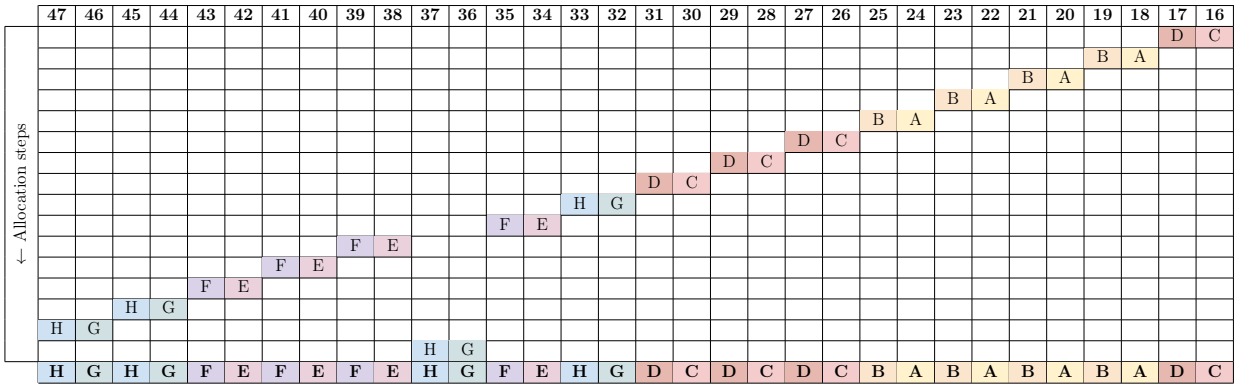
Across both groups there are three sets of sixteen consecutive Stage 2 nodes: [0, 15], [16, 31] and [32, 47]. Within each of these groups there is a clear concentration of connections from the central nodes to the two pairs of BLEs (A, B) and (E, F). These are flanked by connections to the pair (D, C), whose inputs come from a nearby set of drivers that are close but not as close, and finally to the pair (H, G), which is similarly allocated in group D but which has the widest range of drivers from group C.

3.4.3.3 Algorithm

Sketches of the algorithms devised to reproduce the UltraScale connection patterns are shown in Algorithms 3, 4 and 5. As usual, the complete algorithms are available in `interconnect_generator/iib_2_to_luts.py` in the BFG source repository.

Group A and B (Alg. 3) allocation is straightforward and follows the description in the previous section. First, half the bounce muxes are assigned to LUTs at even offsets from the

Table 3.10: IIB Stage 2 to BLE connections, group D



start of the LUT range. The remaining half are assigned to even offsets from the end of the range. The same is done for bypass muxes in excess of the number of LUTs, except that now the halves are allocated to *odd* offsets from the front and back instead. These excess bypass muxes are termed the "shoulder". Finally, a bypass mux from the non-shoulder group is assigned in sequence to each LUT.

Group C is more interesting because, if nothing else, LUTs seem to be allocated together. There also appear to be two places where two of the same LUT pairs end up driven by adjacent muxes, around Stage 2 nodes 2-5 and 8-11. The algorithm we present does not explicitly *seek* to replicate this property, but does so nonetheless. Such is the precarity of generalising from a single example: perhaps we do not capture the initial design intent, and will not be able to translate it to different parameter sets. Perhaps (though it is unlikely), the original designers did the same thing. In any case, our algorithm works by allocating successive pairs of LUTs to pairs of Stage 2 drivers in strides, then changing the starting place for the allocation. We search for a place to start strides in alternating directions, but we always allocate pairs within a stride in the same direction. Each LUT pair is allocated twice as part of two successive strides before we move on to the next set of pairs. The algorithm (Alg. 4), whose output allocation order matches the rows of Table 3.9, proceeds as follows:

(1) Put the LUTs A-H in pairs, T. We will track an index over the LUT pairs j, a search position in the list of Stage 2 drivers k, a search direction δ, and a stride counter q.

(2) Start somewhere around the middle of the input nodes $k \leftarrow 8$ and in the reverse direction $\delta \leftarrow -1$.

(3) Allocate the LUT pair T_{j+q} to the drivers k and $k+1$. Increment q by 1. Move k by 2 in the current direction, $k \leftarrow k + 2\delta$. Track the minimum and maximum k seen so far.

(4) Repeat (3) for $q \in (0, 1)$. The stride length is the number of consecutive pairs we allocate before changing our starting position in the the list of pairs, j.

(5) Jump k to whichever of $k_{\min}$ or $k_{\max}$ is in the current direction, then switch direction.

(6) Go to (3) again and repeat the strides from the new starting position, until all LUT pairs have been allocated or no drivers are available.

Algorithm 3 Generate IIB S2 Group A & B to BLE connections

$\mathcal{B}, \mathcal{P} \leftarrow$ bounce, bypass muxes; $\mathcal{L} \leftarrow$ LUTs
for $i \in \{0, ..., \lfloor |\mathcal{B}|/2 \rfloor - 1\}$ **do**
    Connect($\mathcal{L}_{2i}$.bypass, $\mathcal{B}_i$) ▷ allocate half of the bounce muxes to start/end bypass inputs
    Connect($\mathcal{L}_{|\mathcal{L}|-1-2i}$.bypass, $\mathcal{B}_{|\mathcal{B}|-1-i}$)
end for
$s \leftarrow \max(|\mathcal{P}| - |\mathcal{L}|, 0)$ ▷ find the size of the shoulder, might be zero
for $i \in \{0, ..., {}^{s}/_{2} - 1\}$ **do**
    Connect($\mathcal{L}_{2i+1}$.bypass, $\mathcal{P}_i$) ▷ assign shoulder $\mathcal{P}_{0:s/2}$ front odd LUTs
    Connect($\mathcal{L}_{|\mathcal{L}|-1-(2i+1)}$.bypass, $\mathcal{P}_{|\mathcal{P}|-1-i}$) ▷ remaining shoulder → odd LUTs at back
end for
for $i \in \{0, ..., |\mathcal{P}| - s\}; j = 0$ to $|\mathcal{L}| - 1$ **do**
    while $\mathcal{L}_j$.bypass is already allocated **do**
        $j \leftarrow j + 1$ ▷ skip
    end while
    Connect($\mathcal{L}_j$.bypass, $\mathcal{P}_{s/2+i}$) ▷ assign remaining bypass muxes to free LUT inputs
end for

The algorithm for group D (Alg. 5) is simpler than that of group C. LUTs are still allocated to Stage 2 drivers in pairs and pairs are still allocated in strides, but now we parameterise the direction of the stride itself and the number of times each pair is repeatedly assigned in a row. A rotation mechanism is added so that Stage 2 drivers can be allocated cyclically over the same range from a shifted starting position. We no longer need to track and switch direction after each stride. Instead, we split the allocation into two phases statically. We also split the list of pairs and the list stage 2 drivers in half, and allocate each half of the drivers to one half of the LUTs.

In the first phase, input pairs are allocated in reverse order to Stage 2 drivers in regular order, once each. In the second phase, input pairs are allocated to the remaining Stage 2 drivers in regular order, three times each. A rotation mechanism shifts the starting position within the remaining range and wraps allocations back to the beginning. The resulting allocation order for group D is shown in Table 5. It is this rotation mechanism that is responsible for the odd-seeming allocation of (G, H):

1. First, pairs (A, B) and (C, D) are allocated across Stage 2 drivers 16 – 31.

    a) In the first phase, (C, D) and (A, B) are allocated to 16 – 19.

Algorithm 4 Generate IIB S2 Group C to BLE connections

$\mathcal{C} \leftarrow$ muxes in group C; $\mathcal{L} \leftarrow$ LUTs
$K \leftarrow 6$ ▷ number of regular inputs per LUT
$n_C \leftarrow \lfloor K|\mathcal{C}|/(|\mathcal{C}| + |D|)\rfloor$ ▷ 2, the number of LUT pins assigned to group C
$T \leftarrow [(\mathcal{L}_{2i}, \mathcal{L}_{2i+1})]_i$ ▷ partition LUTs into consecutive pairs
$A \leftarrow [\perp]^{|\mathcal{C}|}$; $k_0 \leftarrow \lfloor|\mathcal{C}|/2\rfloor$; $\delta \leftarrow -1$, $j \leftarrow 0$
while $\perp \in A$ **do**
  $k_{\min}, k_{\max} \leftarrow k_0$
  for $p = 1, \ldots, n_C$ **do**
    $k \leftarrow k_0$; $q \leftarrow 0$
    while $q < 2$ **do**
      if $A_k = \perp$ **then**
        $A_k, A_{k+1} \leftarrow T_{j+q}$ ▷ unpack and allocate the pair
        $q \leftarrow q + 1$
      end if
      $k \leftarrow (k + 2\delta) \bmod |\mathcal{C}|$; $k_{\min} \leftarrow \min(k_{\min}, k)$; $k_{\max} \leftarrow \max(k_{\max}, k)$
    end while
    $\delta \leftarrow -\delta$
    $j \leftarrow j + 2$
    $k_0 \leftarrow \begin{cases} k_{\min} & \delta < 0 \\ k_{\max} & \delta > 0 \end{cases}$
  end for
end while
for i in $\{0, ..., |A| - 1\}$ **do** ▷ execute allocations
  Connect($\mathcal{C}_i$, A_i) ▷ connect to first available input
end for

b) In the second phase, (A, B) is allocated three times, then (C, D) is allocated three times, covering the range of the remaining Stage 2 drivers 20 – 31.

2. Second, pairs (E, F) and (G, H) are allocated across Stage 2 drivers 32 – 47.

   a) This proceeds as in the first case, except that in the second the starting position is offset within the range by 1. (E, F) is allocated three times starting at #38, forcing a wrap around and the final (H, G) allocation we see to (36, 37).

3.5 Output Interconnect Block

The Output Interconnect Block (OIB) is responsible for the distribution of BLE outputs to outgoing interconnect wiring and back to the inputs of the BLEs as feedback. It is

Algorithm 5 Generate IIB S2 Group D to BLE connections

$\mathcal{D} \leftarrow$ muxes in group D; $\mathcal{L} \leftarrow$ LUTs
$m \leftarrow |\mathcal{D}|$; $r \leftarrow m/(2|T|)$; $A \leftarrow [\perp]^m$
$f_{\max} \leftarrow$ max fan-out of Stage 2 drivers
$T \leftarrow [(\mathcal{L}_{2i}, \mathcal{L}_{2i+1})]_i$ ▷ partition LUTs into consecutive pairs
for $h \in \{0, 1\}$ **do**
  $T^{(h)} \leftarrow T\left[h\frac{|T|}{2} : (h{+}1)\frac{|T|}{2}\right]$, $a \leftarrow hm/2$, $i \leftarrow a$
  ▷ place tuples of half h in reverse order, sequentially
  for $k = 0, \ldots, |T|/2 - 1$ **do**
    $k' \leftarrow \left[\frac{|T|}{2} - 1 - k\right]$
    $A_i, A_{i+1} \leftarrow T^{(h)}_{k'}$
    $i \leftarrow i + 2$
  end for
  $b \leftarrow i$, $n \leftarrow m/2 - |T|$
  ▷ place each tuple of half h exactly $r{-}1$ times at cyclically rotated positions within $A[b : b{+}n)$
  for $j' = 0, \ldots, |T|/2 - 1$ **do**
    for $p = 0, \ldots, r - 2$ **do**
      $b' \leftarrow b + (2j'(r{-}1) + 2p + 2h) \bmod n$
      $A_{b'}, A_{b'+1 \bmod n} \leftarrow T^{(h)}_{j'}$
    end for
    $i \leftarrow i + 2(r{-}1)$
  end for
end for
for $i \leftarrow 0, \ldots, m - 1$ **do** ▷ execute allocations
  if $A_i \neq \perp$ **then**
    Connect($\mathcal{D}_i$, A_i.input) ▷ connect to first available input
  end if
end for
  ▷ Greedy residual: pair muxes with remaining fan-out to LUTs with free inputs, in ordinal order; repeat until no pairs remain
while $\exists\, \mu \in \mathcal{D}$ with fanout(μ) $< f_{\max}$, $\exists\, L \in \mathcal{L}$ with free input **do**
  for all $(\mu, L) \in \text{zip}(\{\mu : \text{fanout}(\mu) < f_{\max}\}, \{L : \text{free}(L)\})$ **do**
    Connect(μ, L.input)
  end for
end while

also responsible for redistributing some incoming interconnect wiring back out to outgoing interconnect wiring. Its structure, like that of the IIB, is a 2-fly Butterfly-ish network.

We number the OIB stages in reverse to the IIB. The *second* stage is closest to the BLEs, and the first stage is closest to the interconnect wiring. The second stage lands (1) all of the BLE outputs and (2) incoming interconnect wiring of all lengths. Again, like the IIB, the network is mirrored on both sides of the tile, so we will limit our analysis to the eastern slice. There are 96× logical nodes per slice, which are in fact 4:1 multiplexers, implemented physically as 48× 5:2 multiplexer structures. Most Stage 2 nodes drive Stage 1 nodes directly and are named INT_NODE_SDQ_x_INT_OUTy, per Table 3.1.

The space of ordinals allocated to this stage is shared by both sides of the tile: $x \in [0, 48)$ in the east, and $x \in [48, 96)$ in the west. This is the first hint we have that the OIB is a shared structure, perhaps situated centrally to both sides of the tile. (Recall that the ordinals count *physical* multiplexer structures, whose two logical halves are distinguished by the y parameter. In total there are $48 \times 2 \times 2 = 192$ logical Stage 2 nodes per tile.)

Like in Stage 1 of the IIB, Stage 2 of the OIB also contains five "non-local" bounce nodes named INT/SDQNODE_{E,W}_x per slice that connect to OIB Stage 1 drivers both in the current tile and in the neighbouring tiles to the north and south. They are numbered $x \in \{0, 2, 91, 93, 95\}$. Once again we find that the regular node set is conspicuously missing logical multiplexers, in this case the five Stage 2 nodes $0_A, 11_A, 45_B, 46_B$ and 47_B (Fig. 3.23). By checking for shared inputs, we again find that the non-local bounce nodes, are in, fact the missing logical multiplexers. The pair SDQNODE_E_2_FT1 and INT_NODE_SDQ_11_INT_OUT1, for example, share 3 of their 4 inputs (see Fig. 3.20).

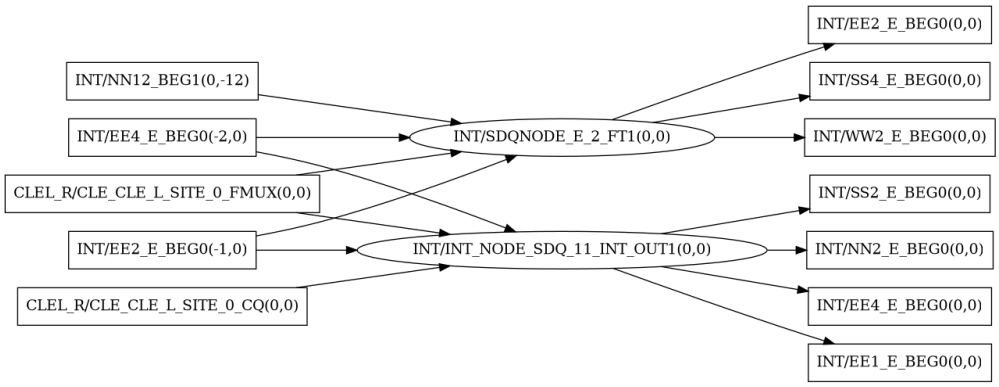


Figure 3.20: A non-local bounce node using the other logical side of a regular Stage 2 multiplexer

The full correspondence betweeen non-local bounce nodes and logical Stage 2 multiplexers is listed in Table 3.11. Although their ordinals do not correspond under a naive mapping (such as $i_A \mapsto 2i$, $i_B \mapsto 2i + 1$), they do coincide roughly to the same ends of the range. This is consistent with our hypothesis of physically-correspondent ordinals: non-local nodes

Table 3.11: Correspondence of non-local OIB Stage 2 nodes to regular OIB Stage 2 nodes

Non-local bounce node	... is the other side of node
SDQNODE_E_0_FT1	INT_NODE_SDQ_0_INT_OUT1
SDQNODE_E_2_FT1	INT_NODE_SDQ_11_INT_OUT1
SDQNODE_E_91_FT0	INT_NODE_SDQ_45_INT_OUT0
SDQNODE_E_93_FT0	INT_NODE_SDQ_46_INT_OUT0
SDQNODE_E_95_FT0	INT_NODE_SDQ_47_INT_OUT0

drive neighbouring slices and would thus be situated near the edges of the physical block to reduce wire length.

The first stage of the OIB is made up mostly of drivers for the interconnect wiring we have already seen. These nodes drive wiring all the way to their destination tiles, at which point their connectivity mirrors the interconnect fan-in we see in our tile. There are 88 wire drivers per side[3]:

- 8× each of lengths 2 and 4, from each of the four cardinal directions, make $8 \times 2 \times 4 = 64$.
- 8× each of length 1 from the north, south and *either* east or west (depending on the side of the tile) make $8 \times 3 = 24$.

Like in Stage 2, OIB Stage 1 also contains 15× logical "bounce" muxes per side. These are named INT/INT_INT_SDQ_x_OUTy, and connect Stage 2:

- back to Stage 2 within the current slice and in the other slice;
- to regular IIB Stage 1 muxes on the input side;
- to the IIB Stage 1 bounce muxes on the input side that also drive IIB Stage 2 in vertical neighbours; and
- to global wire drivers.

These connections provide path diversity by shuffling between Stage 2 nodes across both sides of the tile.

3.5.1 Interconnect to OIB Stage 2

It is tempting to start our treatment of Stage 2 with the fan-out from the BLEs, which are central to the tile's logic. As we shall see in the next section, that would be a mistake. Let

[3]In the overview shown in Figure 3.3 we listed 104 OIB Stage 1 nodes. This is because we included half of the interconnect crossbar, which is shared by both slices. It has 8 wires leaving in 4 directions for $8 \times 4 = 32$ drivers, so we figured $88 + {}^{32}/_{2} = 104$.

us instead consider how the incoming interconnect wiring drives Stage 2 of the OIB: Figure 3.21 shows the adjacency graph of incoming interconnect wiring over Stage 2 multiplexers. Unlike the interconnect wiring which drove Stage 1 of the *Input* Interconnect Block, where only wires of length 1 and 2 were present, here we find connections from all interconnect lengths in each of the cardinal directions, with one exception.

Just as we saw in fan-in to Stage 1 of the input block, wires of length-1 travelling *east* only drive OIB Stage 2 multiplexers on the *west* side of the tile, and vice versa. As we noted in Section 3.4, this makes a good deal of physical sense, since in each case the source of the length-1 wire is an immediately adjacent neighbour.

In fact, there are *no* length-1 east-faring wires from the western slices or length-1 west-faring wires from the eastern slices. It is not clear why this is, but a reasonable explanation is that doing so simply keeps the delay of length-1 wiring more uniform. Consider that in the UltraScale scheme, all length-1 wires travel only to immediate neighbours, including in the vertical directions. A length-1 wire from the western slice to the western or eastern slice on the *eastern* neighbour would cross 1 or 2 (respectively) extra slices when compared to the other length-1 wires and, as a result, would have a significantly higher delay.

Again as we saw in the input block, interconnect driving OIB Stage 2 exhibits a remarkably consistent pattern $\lambda^{(d)}$ for each incoming interconnect wiring direction $d \in \{N, E, S, W\}$. Three instances each of the patterns for the north $\lambda^{(N)}$ and south $\lambda^{(S)}$ wiring are drawn in boxes on Figure 3.21. Here, too, we note special treatment of a special set of nodes $\sigma_j^{(O)} = \{11, 22, 33, 44\}$.

3.5.1.1 Algorithm

The algorithm proceeds just as it did for interconnect fan-in to Stage 1 of the IIB (Section 3.4.1), but with different template patterns and starting parameters. Tables 3.12b - 3.12d provide synopses for each of the OIB Stage 2 patterns $\lambda^{(d)} : d \in \{N, E, S, W\}$. The full code can be found in `interconnect_generator/luts_to_oib_2.py` in the BFG source respository. Table 3.14 provides the initial parameters for each pattern. The final clean up phases still differ slightly between patterns, so the case for $\lambda^{(S)}$ is provided as an example in Table 3.13. Again, the driving interconnect wires are tracked by index i and the OIB Stage 2 nodes are tracked by index j. At each step i and/or j are incremented, and some subset of the incoming wire sizes (in the pattern direction) are connected.

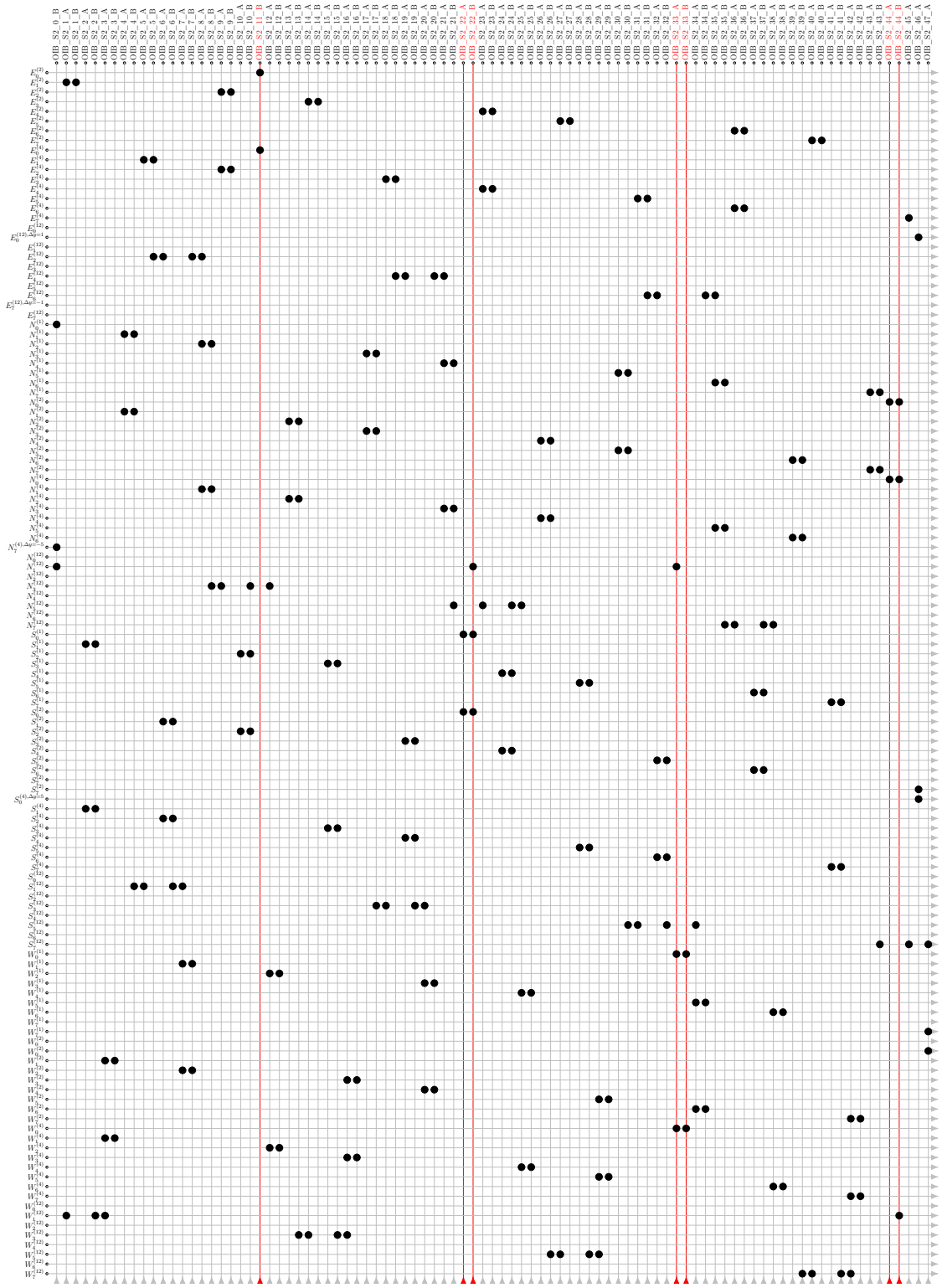


Figure 3.21: Interconnect to OIB S2

Table 3.12: Synopsis of algorithmic steps when connecting interconnect wiring to OIB Stage 2 multiplexers

(a) North wires, $\lambda^{(N)}$

	Phases			
	j	$j \leftarrow \Upsilon(j,4)$	$j \leftarrow \Upsilon(j,2)$	$j \leftarrow \Upsilon(j,2)$
$N^{(1)}$	$i \rightarrow j_A, j_B$	$i+1 \rightarrow j_A, j_B$		
$N^{(2)}$	$i \rightarrow j_A, j_B$			$i+1 \rightarrow j_A, j_B$
$N^{(4)}$		$i \rightarrow j_A, j_B$		$i+1 \rightarrow j_A, j_B$
$N^{(12)}$		$i+2 \rightarrow j_B, (j+1)_A$	$i+2 \rightarrow j_B, (j+1)_A$	

(b) East wires, $\lambda^{(E)}$

	j	$j \leftarrow \Upsilon(j,4)$	$j \leftarrow \Upsilon(j,2)$	$j \leftarrow \Upsilon(j,2)$
$E^{(1)}$	$i \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$		
$E^{(2)}$	$(i+1) \rightarrow j_A, j_B$			$(i+2) \rightarrow j_A, j_B$
$E^{(4)}$		$(i+1) \rightarrow j_A, j_B$		$(i+2) \rightarrow j_A, j_B$
$E^{(12)}$		$(i+1) \rightarrow j_B, (j+1)_A$	$(i+1) \rightarrow j_B, (j+1)_A$	

(c) South wires, $\lambda^{(S)}$

	j	$j \leftarrow \Upsilon(j,2)$	$j \leftarrow \Upsilon(j,2)$	$j \leftarrow \Upsilon(j,4)$
$S^{(1)}$	$i \rightarrow j_A, j_B$			$(i+1) \rightarrow j_A, j_B$
$S^{(2)}$			$i \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$
$S^{(4)}$	$i \rightarrow j_A, j_B$		$(i+1) \rightarrow j_A, j_B$	
$S^{(12)}$		$i \rightarrow j_B, (j+1)_A$	$i \rightarrow j_B, (j+1)_A$	

(d) West wires, $\lambda^{(W)}$

	j	$j \leftarrow \Upsilon(j,2)$	$j \leftarrow \Upsilon(j,5)$	$j \leftarrow \Upsilon(j,4)$
$W^{(1)}$			$i \rightarrow j_A, j_B$	$(i+1) \rightarrow j_A, j_B$
$W^{(2)}$		$i \rightarrow (j+1)_A, (j+1)_B$	$(i+1) \rightarrow j_A, j_B$	
$W^{(4)}$		$i \rightarrow (j+1)_A, (j+1)_B$		$(i+1) \rightarrow j_A, j_B$
$W^{(12)}$	$i \rightarrow j_B, (j+1)_A$	$i \rightarrow j_B, (j+1)_A$		

Table 3.13: Final clean-up phase for south wires, $S^{(i)} : i \in 1, 2, 4, 12$, after completing $\lambda^{(S)}$

	j	$j \leftarrow \Upsilon(j, 4)$
$S^{(1)}$	$i \to j_A, j_B$	
$S^{(2)}$		$i \to (j+1)_A$
$S^{(4)}$	$i \to j_A, j_B$	$(i+1) \to (j+1)_A$
$S^{(12)}$		$i \to (j-2)_B, j_A, (j+2)_A$

Table 3.14: Pre- and post-pattern parameters for $\rho^{(d)} : d \in \{N, E, S, W\}$

	i_0	j_0	Δi	Δj
North, $\lambda^{(N)}$	1	4	2	4
East, $\lambda^{(E)}$	1	1	2	4
South, $\lambda^{(S)}$	1	2	2	4
West, $\lambda^{(W)}$	1	0	2	1

Unlike in the IIB, we make the treatment of special indices $j \in \sigma_j^{(O)}$ explicit in the OIB case. We define a function Υ (Alg. 6) to increment j by some minimum value each time, skipping over special indices. The increments to j *between* phases must all explicitly skip special indices and therefore must be performed through Υ. However, there are now additional cases *within* each phase where we connect to an offset of j, denoted for example by $(j+1)$. These offsets do *not* skip special indices, leading to some spurious connections to special nodes. These spurious inputs will be broken and re-distributed to their neighbours later. It is not clear why the treatment of special nodes is inconsistent between the two cases (or even that it really is different), but it nonetheless reproduces what we see in the UltraScale.

Algorithm 6 Increment x by some minimum δ, skipping members of the special set $\sigma_j^{(O)}$

```
function Υ(x, δ)
    x' ← x + δ                                  ▷ the minimum final value
    δ' ← δ                                      ▷ copy for modification
    for s ∈ σ_j^(O) do
        if s > x and s ≤ x' then                ▷ if a special index falls within the range, add 1
            δ' ← δ' + 1
        end if
    end for
    return x + δ'
end function
```

Table 3.15: Connections from interconnect to special OIB Stage 2 nodes

Wire direction, size	Source side	Wire index, i	Destination side	Stage 2 mux, j
$N^{(2)}$	East	0	East	$44_A, 44_B$
$N^{(4)}$	East	0	East	$44_A, 44_B$
$E^{(1)}$	East	0	West	$33_A, 33_B$
$E^{(2)}$	East	0	East	11_B
$E^{(4)}$	East	0	East	11_B
$S^{(1)}$	East	0	East	$22_A, 22_B$
$S^{(2)}$	East	0	East	$22_A, 22_B$
$W^{(4)}$	East	0	East	$33_A, 33_B$
$N^{(2)}$	West	0	West	$44_A, 44_B$
$N^{(4)}$	West	0	West	$44_A, 44_B$
$E^{(2)}$	West	0	West	11_B
$E^{(4)}$	West	0	West	11_B
$S^{(1)}$	West	0	West	$22_A, 22_B$
$S^{(2)}$	West	0	West	$22_A, 22_B$
$W^{(1)}$	West	0	East	$33_A, 33_B$
$W^{(4)}$	West	0	West	$33_A, 33_B$
$N^{(12)}$		0	West	$22_B, 33_A$
$N^{(12)}$		1	East	$22_B, 33_A$
$W^{(12)}$		0	West	44_B
$W^{(12)}$		1	East	44_B

In the final phase of the algorithm, those spurious connections to special nodes are moved to the first available multiplexer earlier or later in the node sequence. Special nodes receive statically-defined connections according to Table 3.15. These connections are more complex than in the Stage 1 case because they include all sizes of interconnect wiring {1, 2, 4, 12}, not just {1, 2}. The connections do not follow a clear pattern, but they do demonstrate some notable properties:

- With the exception of connections heading south, wires of different sizes in each direction are spread out across the four special nodes in $\sigma_j^{(O)}$. Since these nodes are equally spaced and span the range of Stage 2 multiplexers, this seems to be a good strategy for ensuring path diversity and routability.
- Length-1 east- or west-faring wires again only land on the nearest side of the tile to their source.
- Only north- and west-faring long wires (length-12) land in Stage 2 of the OIB.

Since the ultimate goal of developing these algorithms is to resize the networks for other configurations, it is troublesome that both the IIB Stage 1 and OIB Stage 2 interconnect fan-in patterns ($\rho^{(d)}$ and $\lambda^{(d)}$) are specified in terms of fixed spans of driven nodes j. An inherent balance needs to be achieved between the number of driving nodes and the number of inputs, and it is expected that some of the benefit of these patterns will be lost if they are stamped out without regard to that. One can easily imagine some configuration of interconnect wiring (across size and number), together with some number of Stage 2 nodes, that yields an incomplete network under these patterns.

Ultimately, we seek to understand *why* these patterns work. That is, we seek to understand and reproduce the properties of the network that are desirable rather than the specific patterns themselves. To that end, at least two properties have emerged consistently:

1. Incoming interconnect bundles are spread out across the full range of nodes, but *each wire* is only connected to pairs of inputs on adjacent nodes, and in pairs of these pairs to nearby nodes. This balances additional path diversity for the overall bundle of (8) interconnect wires with the wire delay of each wire.

2. Incoming interconnect from all directions are interleaved to balance optionality in each direction, but there is also some bias. The clearest such bias is for vertical and eastward connections.

In fact, it seems that a stochastic approach that preserves these invariants might be just as effective for generating the connection pattern. Perhaps that is where the UltraScale designers started.

In the meantime, we can employ practical mitigations to enforce these invariants around ill-fitting static patterns. For example, a post-pattern phase reassigns over-allocated nodes to free inputs nearby after the patterns are exhausted, enforcing fan-in and fan-out bounds while trying to preserve property (1) above.

3.5.2 BLE outputs to Stage 2

We can now consider how the various BLE outputs drive multiplexers in Stage 2 of the OIB. Unfortunately, we could not discern a clear pattern from the adjacency matrix for this section (Fig. 3.22). So we opted for a different approach. We start with several observed properties of the network, all of which we have already seen in some form:

1. As we saw in earlier stages, outputs usually drive clusters of nearby inputs. In this case, though, we can be more specific:

    a) 24 of 32 (conveniently ¾) outputs drive a shared input on each of two muxes. Of these,

    - 22 drive muxes spaced 2-3 apart, e.g. muxes (39, 41), or (9, 12); and
    - 2 drive the familiar special ordinals, (0, 22) and (11, 33).

   b) 8 of 32 (conveniently ¼)outputs drive the unique inputs for four consecutive logical muxes, with one exception being a single output that drives four special indices (11, 22, 33, 44). Half of these are from the BLE "MUX" outputs. The pattern of connection to successive unique inputs is always (B, A, B, A).

2. Of the 48 physical OIB Stage 2 muxes, five have only one of their two logical halves driven by BLEs. We call these "untied" muxes. The remainder, which we call "tied" muxes, have both logical halves driven by BLEs.

3. The connections from the BLEs are balanced between the both halves of the output range.

Absent a clear pattern, we opt for a stochastic algorithm that preserves these properties of the network. Because the interconnect fan-in to OIB Stage 2 is rigidly dictated by patterns, using a stochastic method for the remaining BLE outputs has the additional benefit of adapting to whatever is left over. That is why we consider the interconnect fan-in first. In fact, it feels as though the UltraScale designers might have done something similar.

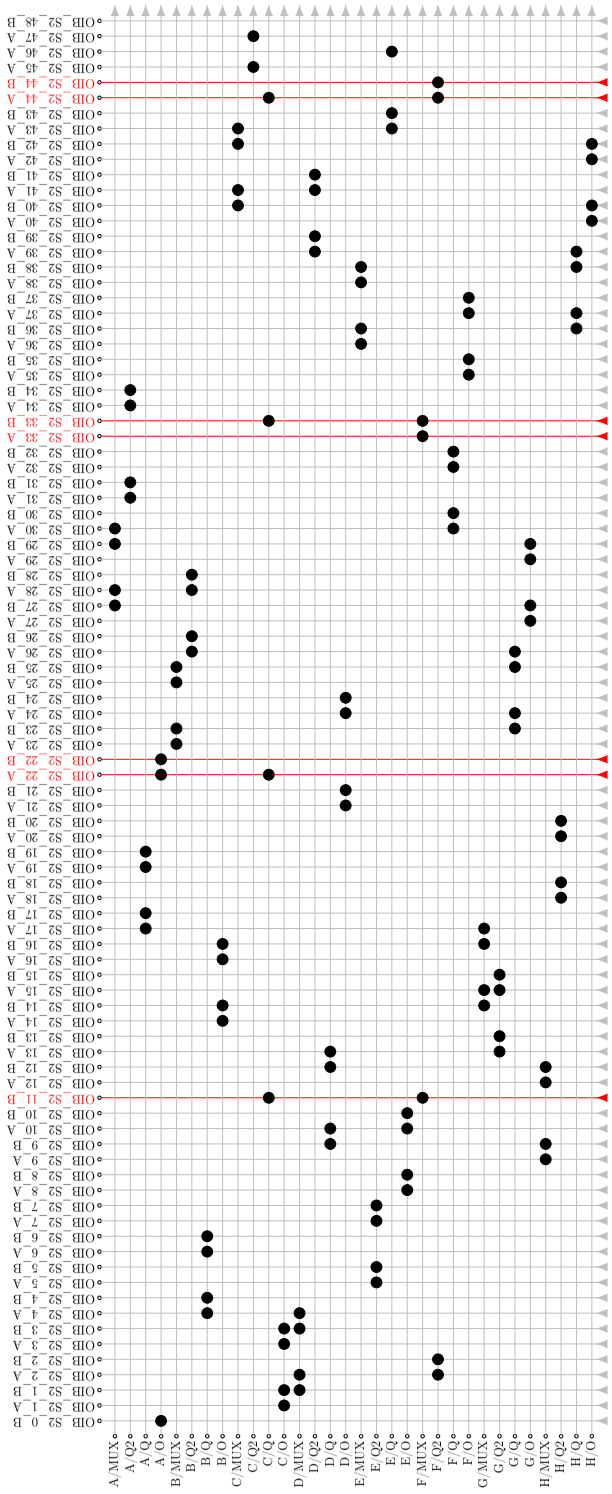


Figure 3.22: BLE outputs to OIB S2

3.5.2.1 Algorithm

Our stochastic approach is, at its simplest, to randomly assign BLE outputs to Stage 2 nodes and then perform fixes as necessary to maintain the noted invariants. The full source is available in `interconnect_generator/luts_to_oib_2.py` in the BFG code repository. The algorithm is split into phases which are each responsible for enforcing a different invariant. A sketch is provided in Algorithm 7, and we enumerate the phases next.

A. Partition the BLE outputs into a set that connect to "untied" multiplexers O_u and a set that connect to "tied" multiplexers O_t, as seen in property (1). Connecting to "tied" multiplexers means connecting to successive A and B sides of two nearby physical muxes. Using j to denote the index receiving (physical) multiplexer and the subscripts A and B to denote its two logical halves, an example of this pattern is to connect to muxes $\{j_A, j_B, (j+2)_A, (j+2)_B\}$. On the other hand, ports in the "untied" set span alternating sides of a consecutive sequence of muxes, such as $\{j_B, (j+1)_A, (j+2)_B, (j+3)_A\}$. (The sequences are actually a function of the fan-out $F = 4$, which dictates the number of connections from each output port.) The two helper methods isFreePairAB and isFreePairBA (Alg. 8) are used to check for the availability of these patterns, as we will see in subsequent phases.

B. Randomly select (again from a uniform distribution) a single untied output $\ell \in O_u$ for connection to the special ordinals $\sigma_j^{(O)}$. The chosen output is removed from the set of untied outputs O_u so that it is not allocated again.

   Then, select another output on the same BLE as ℓ, opting for a registered output if one is available, and connect it to the Stage 2 bounce nodes in the middle (m) and at the end (z) of the range. (As a result of this step, one registered output on the chosen BLE will always be connected to either the special or a bounce multiplexer.) We remove the connected bounce muxes from the available set $\mathcal{B}$. Lastly, we ensure that one instance of each logical output port type {Q, Q2, MUX, O} drives a unique bounce node, and that all bounce nodes are driven. Ports are chosen randomly from different BLEs each time.

C. At this point, connections should be made from two random "tied" ports to subsets of the special indices $\{0, 11\}$ and $\{22, 33\}$ in a similar manner to the special connection from the untied port. This is omitted from the sketch.

D. Connect the remaining untied inputs to the "untied" sequence of nodes starting at random positions within the range of *regular* (non-bounce) Stage 2 devices $\mathcal{O}^{(2)}$. In this step we sample with replacement, so it is possible (but unlikely) that the same Stage 2 mux receive connections from two different inputs.

   Because the interconnect was already allocated to Stage 2 nodes, we expect to encounter muxes with no free inputs by this stage. Stage 2 nodes have two unique inputs

Algorithm 7 Allocate BLE outputs to OIB Stage 2 muxes

$\mathcal{O}^{(2)} \leftarrow$ a list of all OIB Stage 2 non-bounce muxes
$\mathcal{L} \leftarrow$ a list of all BLEs
$M \leftarrow |\mathcal{O}_2|$; $F \leftarrow 4$, the nominal fan-out for each BLE output
$\sigma_j^{(O)} \leftarrow \{11, 22, 33, 44\}$ ▷ a list of the indices of special OIB Stage 2 muxes
$N_u \leftarrow 5$. the number of "untied" muxes, of which only one logical half is driven
$\mathcal{U} \leftarrow \{0, \lfloor M/F \rfloor\} \cup \{M - k : k \in \{1, 2, ..., N_u - 2\}\}$ ▷ indices of untied muxes
$\mathcal{T} \leftarrow \{0, 1, ..., M - 1\} \setminus \mathcal{U}$ ▷ all other indices
$U \leftarrow \left\{ \left(\mathcal{O}_i^{(2)}\right)_A : i \in \mathcal{U} \right\}$ ▷ the A-side ports of all untied muxes
$T_A \leftarrow \left\{ \left(\mathcal{O}_i^{(2)}\right)_A : i \in \mathcal{T} \right\}$; $T_B \leftarrow \left\{ \left(\mathcal{O}_i^{(2)}\right)_B : i \in \mathcal{T} \right\}$ ▷ ports of all tied muxes
$P \leftarrow \{$Q, Q2, MUX, O$\}$ ▷ the output ports on every BLE
$L \leftarrow \{(\mathcal{L}_i)_s : i \in [0, 7], s \in P\}$ ▷ all output BLE ports
$\mathcal{B} \leftarrow$a list of non-local bounce muxes
$n_T \leftarrow |T_A|$;

— **Phase A: Parition LUT outputs into drivers of tied and untied muxes** —
$O_u \leftarrow$ SAMPLE$(L,\ |L|/8)$ ▷ find a random subset of $|L|/8$-many BLE outputs
$O_t \leftarrow L \setminus O_u$

— **Phase B: Allocate specials** —
$Q_B \leftarrow \mathcal{L}$ ▷ to track unbounced BLEs
$Q_P \leftarrow P$ ▷ to track unbounced ports
if $O_u \neq \emptyset$ **then**
    $\ell \leftarrow$ SAMPLE$(O_u, 1)$; $O_u \leftarrow O_u \setminus \{\ell\}$ ▷ grab a random BLE port from the untied set
    $x \leftarrow$ B
    for $j \in \sigma_j^{(O2)}$ **do** ▷ connect to special indices, alternating sides
        CONNECT $\left(\ell, \left(\mathcal{O}_j^{(2)}\right)_x\right)$
        $x \leftarrow$ A **if** $x =$ B **else** A
    end for
    $q \leftarrow$ a registered output on the same BLE as ℓ with $q \neq \ell$
    $m \leftarrow \mathcal{B}_{\lfloor |\mathcal{B}|/2 \rfloor}$; $z \leftarrow \mathcal{B}_{|\mathcal{B}|-1}$
    CONNECT (q, m); CONNECT (q, z)
    $\mathcal{B} \leftarrow \mathcal{B} \setminus \{m, z\}$; $Q_B \leftarrow Q_B \setminus m, z$; $Q_P \leftarrow Q_P \setminus q$
end if
while $\mathcal{B} \neq \emptyset \wedge Q_P \neq \emptyset \wedge \exists q \in P\ \neg$CONNECTED$(x_q, b)\ \forall x \in \mathcal{L}, \forall b \in \mathcal{B}$ **do**
▷ for ports that aren't connected to bounce muxes on any BLE
    $b' \leftarrow \mathcal{B}_0$; $\mathcal{B} \leftarrow \mathcal{B} \setminus \{b'\}$
    $y \leftarrow$ SAMPLE$(Q_B, 1)$; $Q_B \leftarrow Q_B \setminus \{y\}$
    $p \leftarrow$ SAMPLE$(Q_P, 1)$; $Q_P \leftarrow Q_P \setminus \{p\}$
    TRYCONNECT (y_p, b')
end while

— **Phase C: Allocate untied mux inputs** $i_B, (i+1)_A, (i+2)_B, (i+3)_A$ —
for $\ell \in O_u$ **do**
  $i \leftarrow \text{RANDINT}(0,\ n_T)$ ▷ unif. distributed random integer in the range $[0, n_t)$
  $j \leftarrow i$; $s \leftarrow \text{FALSE}$
  while $j - i < F$ **do**
    $k \leftarrow j \bmod n_T$; $\mu \leftarrow (T_A)_k$ **if** s **else** $(T_B)_k$
    $\text{TRYCONNECT}(\ell, \mu)$; $j \leftarrow j + 1$; $s \leftarrow \neg s$
  end while
end for

— **Phase D: Allocate tied mux inputs** $i_A, i_B, (i+1)_A, (i+1)_B$ —
$g_{\min} \leftarrow 1$; $g_{\max} \leftarrow 3$
$Q \leftarrow \text{SHUFFLE}(O_t)$
for $\ell \in Q$ **do**
  $\vec{\mu} \leftarrow \text{FINDFREEMUXPAIR}(g_{\min}, g_{\max}, 0)$
  if $\vec{\mu} = \text{NIL}$ **then**
    $\vec{\mu} \leftarrow \text{FINDFREEMUXPAIR}(g_{\min}, g_{\max}{+}2, 0)$
  end if
  if $\vec{\mu} = \text{NIL}$ **then**
    $\vec{\mu} \leftarrow \text{FINDFREEMUXPAIR}(g_{\min}, g_{\max}{+}2, 0, \text{TRUE})$
  end if
  if $\vec{\mu} = \text{NIL}$ **then break**
  end if
  for $\mu \in \vec{\mu}$ **do**
    $\text{TRYCONNECT}(\ell, \mu)$
  end for
end for

(one per logical side) and three shared inputs: if there is no room for a new connection, we abandon the attempt and reconcile in the final phase.

E. Connect the remaining tied inputs, in a random order, to "tied" sequences of nodes where possible. Shuffle the list of tied ports O_t and then, for each port ℓ, find the first available sequence of Stage 2 nodes that can accommodate the "tied" pattern, falling back to the "untied" pattern if none are available. The helper function FINDFREEMUXPAIR (Alg. 8) searches through all Stage 2 nodes and returns the start of a range with the requisite free inputs to accommodate one of the patterns.

F. Finally, clean up. Try to connect any ports with spare fan-out ($\text{FANOUT}(\ell) < F$ to any Stage 2 nodes with free inputs. Additionally, enforce a set limit on the number of connections to any single Stage 2 node that come from BLEs.

— **Phase E: Clean up** —
$N^{(O2)} \leftarrow 4$ ▷ number of inputs per *logical* Stage 2 mux
$N_L^{(O2)} \leftarrow 2$ ▷ maximum number of inputs per *logical* Stage 2 mux which can be BLEs
$\Theta \leftarrow \{\ell \in L : \text{FANOUT}(\ell) < F\}$
$R \leftarrow U \cup T_A \cup T_B$ ▷ all available mux ports
$\Gamma \leftarrow \{\mu \in R : \text{FANIN}(\mu) < N^{(O2)}\}$
while $\Theta \neq \emptyset \ \wedge \ \Gamma \neq \emptyset$ **do** ▷ connect ports with available fan-out and fan-in
    $\ell \leftarrow \text{POP}(\Theta)$; $\mu \leftarrow \text{POP}(\Gamma)$
    TRYCONNECT(ℓ, μ)
    if FANIN$(\mu) < N^{(O2)}$ **then** PUSH(Γ, μ)
    end if
    if FANOUT$(\ell) < F$ **then** PUSH(Θ, ℓ)
    end if
end while
for $\mu \in R$ **do**
    while $|\text{BLEINPUTS}(\mu)| > N_L^{(O2)}$ **do** ▷ enforce max. number of inputs from BLEs
        DISCONNECT$((\text{BLEINPUTS}(\mu))_0), \mu)$
    end while
end for

Algorithm 8 Helper methods for allocation of BLE outputs to OIB Stage 2 muxes

function ISFREEPAIRAB(k, g)
    return mux inputs $\{k_A, k_B, (k+g+1)_A, (k+g+1)_B\}$ **if** they are all free, **else** NIL
end function

function ISFREEPAIRBA(k, g)
    return mux inputs $\{k_B, (k+1)_A, (k+g)_B, (k+g+1)_A\}$ **if** they are all free, **else** NIL
end function

function FINDFREEMUXPAIR$(g_{\min}, g_{\max}, \sigma, \alpha)$
    for $i \leftarrow 0$ **to** $M-1$ **do**
        $k \leftarrow (\sigma + i) \bmod M$
        for $g \leftarrow g_{\min}$ **to** $g_{\max}$ **do**
            $\vec{\mu} \leftarrow$ ISFREEPAIRAB(k, g) **if** $\neg\alpha$ **else** ISFREEPAIRBA(k, g)
            if $\vec{\mu} \neq$ NIL **then**
                return $\vec{\mu}$
            end if
        end for
    end for
    return NIL
end function

3.5.3 Stage 2 to Stage 1

The fan-out from Stage 2 to Stage 1 of the OIB follows a familiar application of template patterns, just as we saw feeding into Stage 1 of the IIB (Sec. 3.4.1) and into Stage 2 of the OIB (Sec. 3.5.1). Yet, "Stage 1 of the OIB" is just our name for outgoing interconnect wiring. It is striking that all fan-in and fan-out from and to the interconnect are organised with the same regime.

The full connectivity of Stage 2 to Stage 1 OIB nodes (in the eastern slice) is shown in Figure 3.23. Unlike the other examples of patterned fan-out we have seen, only two distinct patterns are needed to describe this network, $\Gamma^{(p)} : p \in \{1, 2\}$, and they are summarised in Table 3.17. We maintain the convention of letting i index the source of each connection (in this case the Stage 2 muxes) and j index the destination of each connection (in this case the Stage 1 muxes). We also reuse the increment function Υ from the fan-in to OIB Stage 2 (Alg. 6), since we again find ourselves needing to increment while ignoring the special Stage 2 indices[4] $i \in \sigma_j^{(O2)}$.

As we have seen elsewhere, the parameters of each pattern are grouped by the direction and size of the interconnect wiring being driven. *Within* such a group, repeated instances n of a pattern share state with the previous instances, again as we have seen before. Examples of instances $\Gamma_n^{(p)}$ are shown in boxes on Figure 3.23. Which pattern applies, and its initial and final parameters, are listed for each of the interconnect bundles $d^{(s)} : d \in \{N, E, S, W\}$, $s \in \{1, 2, 4\}$ in Table 3.18. Length-12 wires do not appear here because they are only driven from the interconnect crossbar (Sec. 3.6).

The two patterns $\Gamma^{(1)}$ and $\Gamma^{(2)}$ are very similar. They both allocate half of the connections to each of two adjacent output nodes, and they both span 13 drivers per instance. Both alternate between the B and A logical halves of each physical mux in the same order. They differ only in the spacing between the central drivers. Their small difference and relative simplicity leads us to notice a bigger problem: the initial values of i and j (respectively i_0 and j_0) are crucial for optimally fitting instances of the patterns together. We have transcribed the initial values for our single example of the UltraScale architecture, but how do we — in general — determine the order of and offsets for each pattern? We will consider this in the next section.

3.5.3.1 Non-local bounce nodes

Lastly, we turn our attention to the non-local bounce nodes from Stage 2, introduced with Section 3.5 and in the fan-out from the BLEs. We know that the 5 non-local bounce nodes are in fact logical multiplexers that have been omitted from the regular list of Stage 2 nodes, so we exclude them from the Stage 2 drivers when allocating patterns Γ. Instead, we consider the allocation of non-local bounce nodes fanning *in* to Stage 1 — from the perspective of Stage 1 nodes — since it also captures which tiles the sources are in.

[4]We initially gave $\sigma^{(O2)}$ the subscript j to indicate that it was a collection of indices that would be referred to by j, but with those nodes now the drivers, it becomes referred to be values i.

Table 3.16: Position and quantity of non-local bounce nodes driving OIB Stage 1

	$i = 0$	$i = 7$
$E_i^{(1)}$	North	Local
$E_i^{(2)}$	Local	
$N_i^{(1)}$		South
$N_i^{(2)}$	North	
$N_i^{(4)}$	2× North	Local
$S_i^{(1)}$		2× South
$S_i^{(4)}$	Local	2× South
$W_i^{(1)}$	North	
$W_i^{(2)}$	Local	2× South
$W_i^{(4)}$		South

Unless otherwise stated, a single driver in each location drives the noted outgoing wire. *North* and *South* indicate that the source non-local bounce nodes sit in the neighbouring tiles to the north and south, respectively. *Local* indicates that the source is in the current tile.

Table 3.16 lists the sources of non-local bounces feeding each of the cardinal directions of outgoing interconnect. Connections are only made to the first ($i = 0$) and last ($i = 7$) wire in a bundle. For example: the last wire in the north-faring wire bundle of length 4 can be driven by one non-local bounce node in the current tile; the last wire in the south-faring bundle of length 4 can be driven by *two* non-local bounce nodes in the tile immediately to the south; the *first* length-1 wire heading East can be driven by a non-local bounce node in the tile immediately north; and so on.

Notably, the last wires are only ever driven by bounce nodes locally or to the south, and the first wires are only every driven locally or from the north. Non-local bounce nodes only connect to vertical neighbours: there are no sources from the east or west. This is another example of biasing vertical bandwidth in the interconnect.

3.5.3.2 Algorithm

A key problem with generalising the pattern templates we have seen here, in Section 3.4.1 and in Section 3.5.1, is to determine their offsets (i_0, j_0). Patterns must be layered such that three three key invariants are preserved:

1. Each Stage 2 driver must connect to exactly $F = 4$ Stage 1 nodes, since that is the nominal fan-out throughout the network.

2. Each Stage 1 6:1 multiplexer must be driven by exactly 6× Stage 2 drivers.

Table 3.17: Fan-out patterns between Stage 2 and Stage 1 of the OIB

(a) Pattern 1, $\Gamma^{(1)}$

Phases			
i	$i \leftarrow \Upsilon(i,4)$	$i \leftarrow \Upsilon(i,1)$ $j \leftarrow j+1$	$i \leftarrow \Upsilon(i,4)$
$i_B, (i+1)_B \rightarrow j$	$i_A, (i+1)_A \rightarrow j$	$i_B, (i+1)_B \rightarrow j$	$i_A, (i+1)_A \rightarrow j$

(b) Pattern 2, $\Gamma^{(2)}$

i	$i \leftarrow \Upsilon(i,3)$	$i \leftarrow \Upsilon(i,2)$ $j \leftarrow j+1$	$i \leftarrow \Upsilon(i,4)$
$i_B, (i+1)_B \rightarrow j$	$i_A, (i+1)_A \rightarrow j$	$i_B, (i+1)_B \rightarrow j$	$i_A, (i+1)_A \rightarrow j$

In each case, i indexes the driving Stage 2 node, j indexes the target Stage 1 node, and $\Upsilon(x,n)$ increments x by at least n while skipping any Stage 2 special indices $\sigma_j^{(O2)}$.

Table 3.18: Initial and final parameters for OIB Stage 2 to Stage 1 pattern instances

Stage 1 Node	Pattern, p	i_0	j_0	Δi	Δj
$N^{(1)}$	1	-5	0	3	1
$N^{(2)}$	1	3	1	3	1
$N^{(4)}$	1	-1	0	3	1
$E^{(1)}$	2	4	1	3	1
$E^{(2)}$	2	0	0	3	1
$E^{(4)}$	2	-4	0	3	1
$S^{(1)}$	1	1	1	3	1
$S^{(2)}$	1	-3	0	3	1
$S^{(4)}$	1	-7	0	3	1
$W^{(1)}$	2			3	1
$W^{(2)}$	2	-6	0	3	1
$W^{(4)}$	2	2	1	3	1

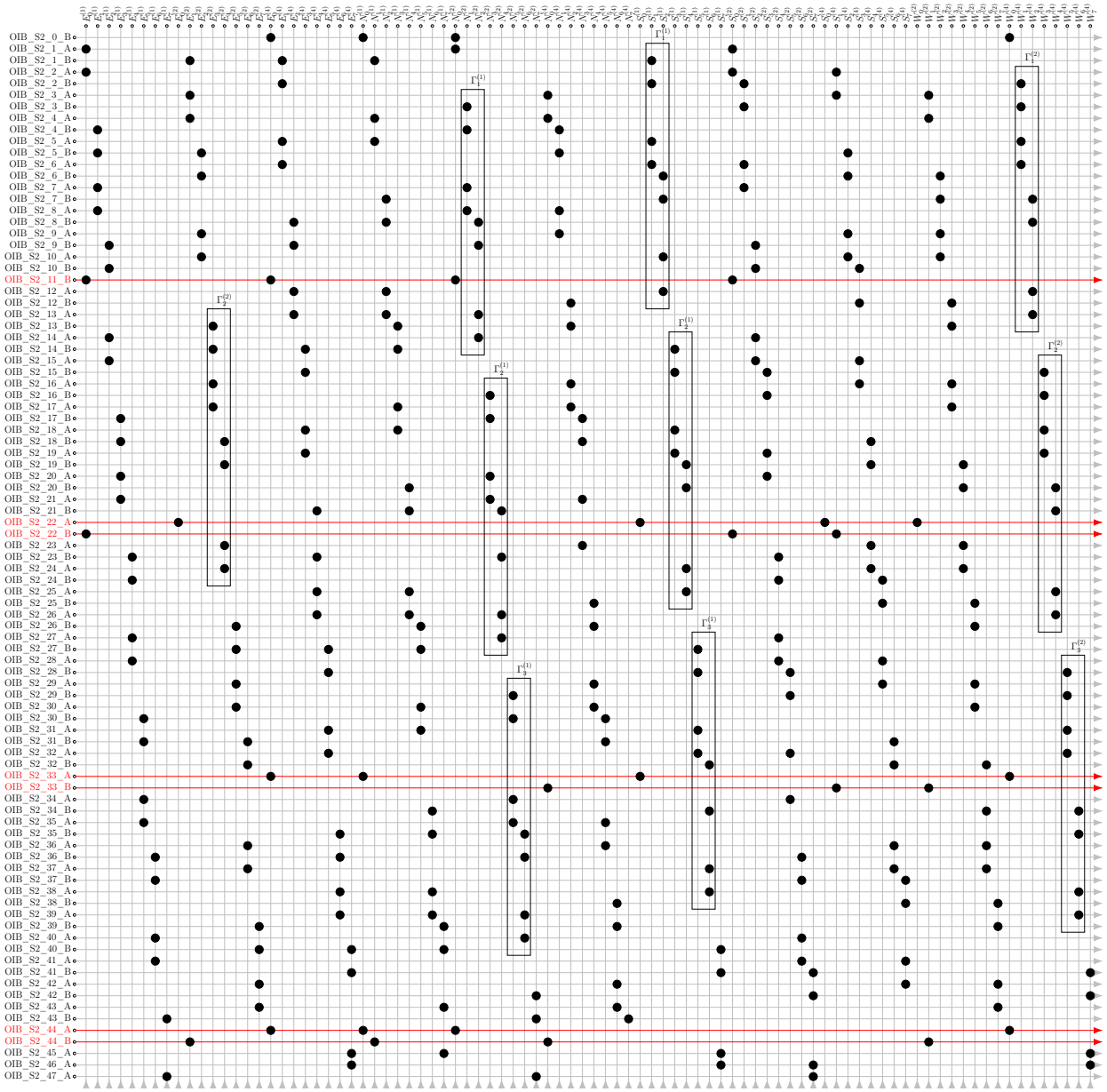


Figure 3.23: OIB S2 to S1

3. Instances of a pattern within a bundle must be applied consecutively to ensure path diversity across the range of Stage 1 and Stage 2 ordinals.

Additionally, the pattern size (i- and j-span) itself may need to change to accommodate different numbers of nodes in Stage 1 and/or Stage 2, and this will effect the required offsets. The general solution to this problem must therefore account for variations in the sets of

input and output nodes, the maximum fan-out F and the size of multiplexers being driven, and is necessary in order resize the network as we would like to do.

For an intuitive sense of what is required, it might help to picture the patterns, as they appear on Figure 3.23, as groups of Tetris blocks falling to the left. The size and shape of the patterns (blocks) might have to change, the blocks within an interconnect bundle can be shifted up or down, but ultimately the blocks must fall (to the left) and leave no gaps. (This is heavily modified Tetris: blocks above or below the grid will be discarded.)

The problem might be framed as bin-packing, except that a) patterns can be truncated at the ends of the range; and b) pattern instances are contiguous, so effectively there are only two shapes. Forcing patterns to be contiguous affords us the property that higher-ordinal Stage 2 nodes drive higher-ordinal Stage 1 nodes, reducing their intervening wire length under our assumption of ordinal-positional correspondence. Ultimately, this is a covering problem whose treatment we leave to future work. Our approach is to avoid it: we blindly apply the patterns Γ using the stated starting parameters (Table 3.18) and then check for violated invariants, shuffling connections to the nearest neighbour as necessary:

1. To each outgoing wire j in each interconnect bundle in Stage 1 of the OIB, connect Stage 2 node i following the applicable pattern and its respective parameters from statically-defined data of Table 3.18.

2. Connect the A and B sides of OIB Stage 2 muxes with special indices to the first available Stage 1 nodes in each interconnect bundle. The set of drivers is that of the $\{A, B\}$ sides of the special indices $\sigma_j^{(O2)}$, except that 11_A is omitted:

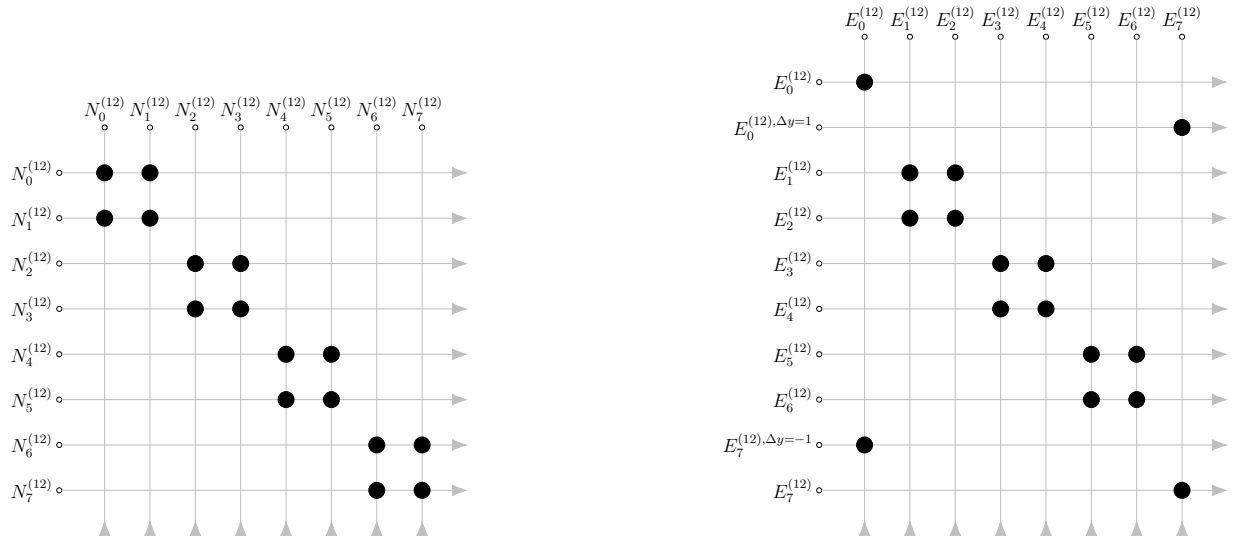
$$\left(\{A, B\} \times \sigma_j^{(O2)}\right) \setminus 11_A$$

   Since the main patterns were applied in (1), we are actually just connecting to any drivers with available inputs.

3. Connect the non-local bounce nodes from Stage 2 according to the statically-defined data of Table 3.16. If a connection is unavailable due to previous allocations in (1) and (2), find the next available input on the next available Stage 1 node.

3.6 Interconnect Crossbar

We consider the interconnect crossbar to be logically part of Stage 1 of the OIB because, like the other multiplexers in OIB S1, it is shared by both the East and West slices and because it drives interconnect wiring. However, this subset is the only portion responsible for driving the longest-length wires (length 12). The only inputs to this section are incoming length-4 and length-12 wires. Its connectivity pattern is the most classically recognisable of any of the groups, mirroring traditional switch-box architectures from the literature.



(a) North-to-north connections

West-to-west and south-south connections exhibit the same pattern.

(b) East-to-east connections

Figure 3.24: Connections between length-12 wires in going the same direction

The full adjacency matrix of the interconnect crossbar is shown in Figure 3.27. We will consider sub-regions independently to elucidate common patterns, starting with the connections between incoming and outgoing length-12 wires heading in the same direction, shown in Figure 3.24. In the North, South and West-heading cases the pattern is identical: the incoming 8 tracks are divided into pairs, $(2k, 2k+1)$ for $k \in \{0, 1, 2, 3\}$ and tracks can connect to either track within their pair. As with traditional crossbars this means that, while track switching is possible, tracks can only switch to other track within their pair.

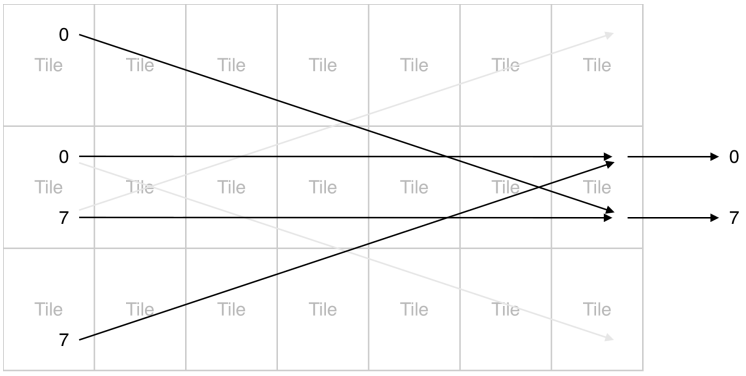
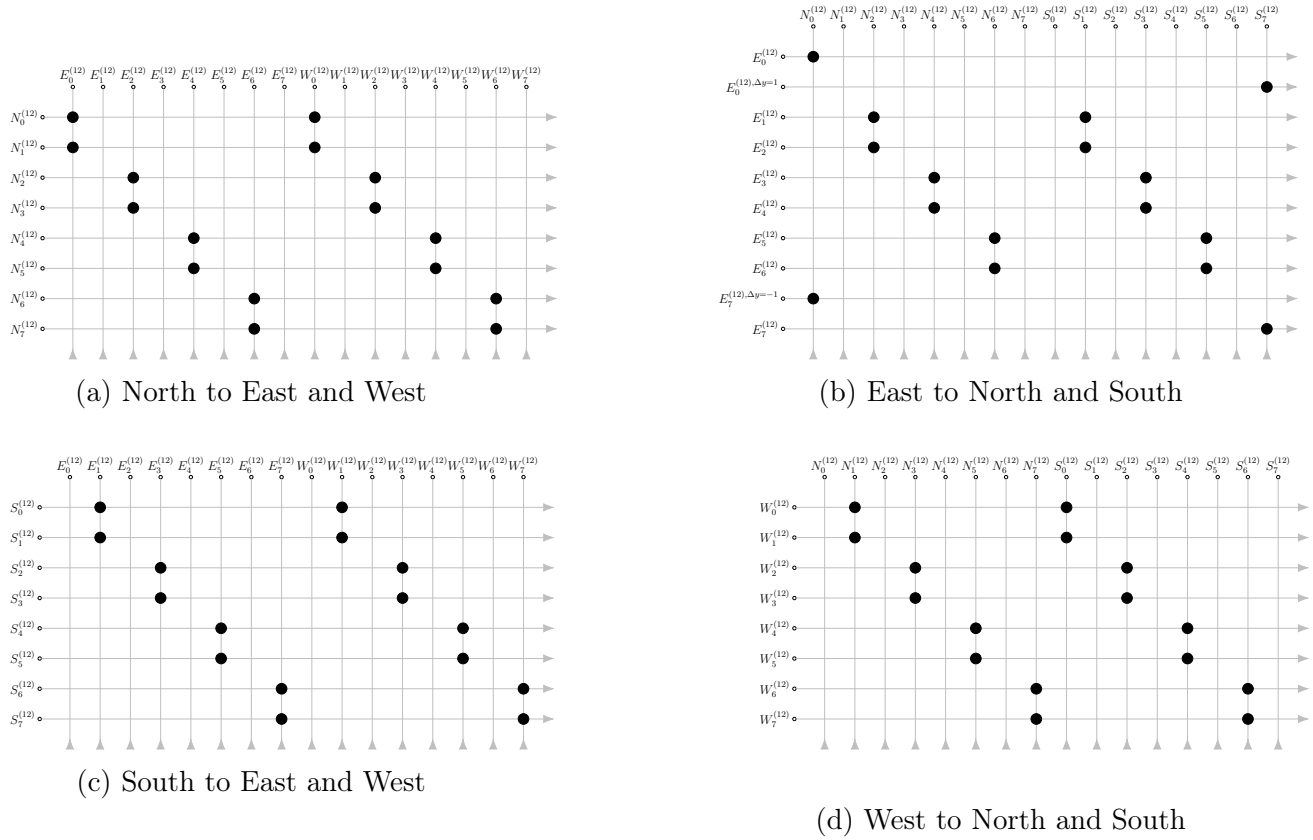


Figure 3.25: Eastward length-12 interconnect wires can travel diagonally up and down



(a) North to East and West

(b) East to North and South

(c) South to East and West

(d) West to North and South

Figure 3.26: Connections between length-12 wires going in different directions

The east-travelling wires (Fig. 3.24b) are similar for $k = \{1, 2, 3\}$, so the pairings are $(1, 2)$, $(3, 4)$ and $(5, 6)$. However, we see that the first group $(7, 0)$ is additionally fed by wires from source tiles one row *above* and one row *below* the current one. Outgoing eastward length-12 track 0 out can be fed by the incoming eastward length-12 track 0 in *or* outgoing eastward length-12 track 7 from the tile diagonally situated 12 slices west and one slice up. Likewise for outgoing track 7, which can be optionally driven by outgoing track 0 in the tile 12 slices west and one slice *down*. See Figure 3.25 for a pictorial representation. This pattern is consistent with the expected benefits of a columnar architecture: since slices have more connections to their neighbours vertically than horizontally, and are thus set up to share the implementation of larger functions within a column, it seems beneficial to be able to connect to vertically adjacent tiles directly. Additionally, this pattern is not present in the westward direction. Though it is pure conjecture, this is consistent with the designers having assumed, or allowed for, some directionality to a datapath being mapped onto the fabric, from west to east.

Figure 3.26 shows the connectivity between length-12 wires intending to change directions. A long wire can turn left or right, but it can never about-face. Of the eight tracks

Table 3.19: Interconnectivity of length-12 North, South, West wires in the interconnect crossbar

The pair (i, i+1) heading:	... can drive length-12 wire(s) on track(s) (all indices are mod 8):			
	North	East	South	West
North	$\{i, i+1\}$	i		i
South		$i+1$	$\{i, i+1\}$	$i+1$
West	$i+1$		i	$\{i, i+1\}$

going in any direction only half can be connected to by the incoming wires from each orthogonal side. Put another way, long tracks leaving the crossbar alternate in being reachable by turns from the left or right, but they are never reachable by both. The connectivity among length-12 wires is summarised in Table 3.19.

The connectivity between length-4 and length-12 wiring follows familiar patterns, but is even further depopulated over what we have seen so far. When connecting to wires in the same direction, incoming length-4 wiring can only connect to the longer wire on the same track. When changing direction, only half of the outgoing tracks are available, but only half of the incoming tracks can switch to them. Additionally, one track heading north and one track heading south are taken from a tile respectively one row below and one row above where would otherwise be expected. That is, track 0 of the southward length-4 wires comes from the site 5 rows above, not 4, and among the connections from northward length-4 wires, track 7 comes from the site 5 rows below, not 4. The effect of this is to provide another fast path to nearby vertical neighbours, which we know from Sections 3.4 and 3.5 to be highly coupled, and is another example of the column-wise interconnect biasing.

Unlike in length-12 wiring, all smaller-length wires are repeated between the two slices on a tile, and the connectivity is mirrored between them. As a result, each connection from a length-4 wire shown in Figure 3.24 represents two connections in the crossbar. Thus, we observe that overall fan-in to a single output length-12 wire is preserved at 8, suggesting that they are driven by 8:1 multiplexers. The connectivity of length-4 to length-12 wiring is summarised in Table 3.20.

3.7 Control Paths

Control multiplexers drive a single control signal within the BLE complex each and can be configured for input from 8 different global signals, as well as constants like *VCC*, which we suspect is a constant "1". In the example of Figure 3.28, the control signal multiplexer is driving the second clock-enable signal `CKEN1` of the right-side logic slice. There are four such signals as well as four separate clocks on four separate resets.

Table 3.20: Connectivity from length-4 to length-12 wires in the interconnect crossbar

Length-4 track i heading:	... can drive the length-12 wire on track(s) (all indices are mod 8):			
	North	East	South	West
North	$i+1$ for odd i	$i+1$		i for even i
East	i for even i	i	i for odd i	
South		$i-1$ for even i	$i-1$	i for odd i
West	$i+1$ for even i		i for odd i	i

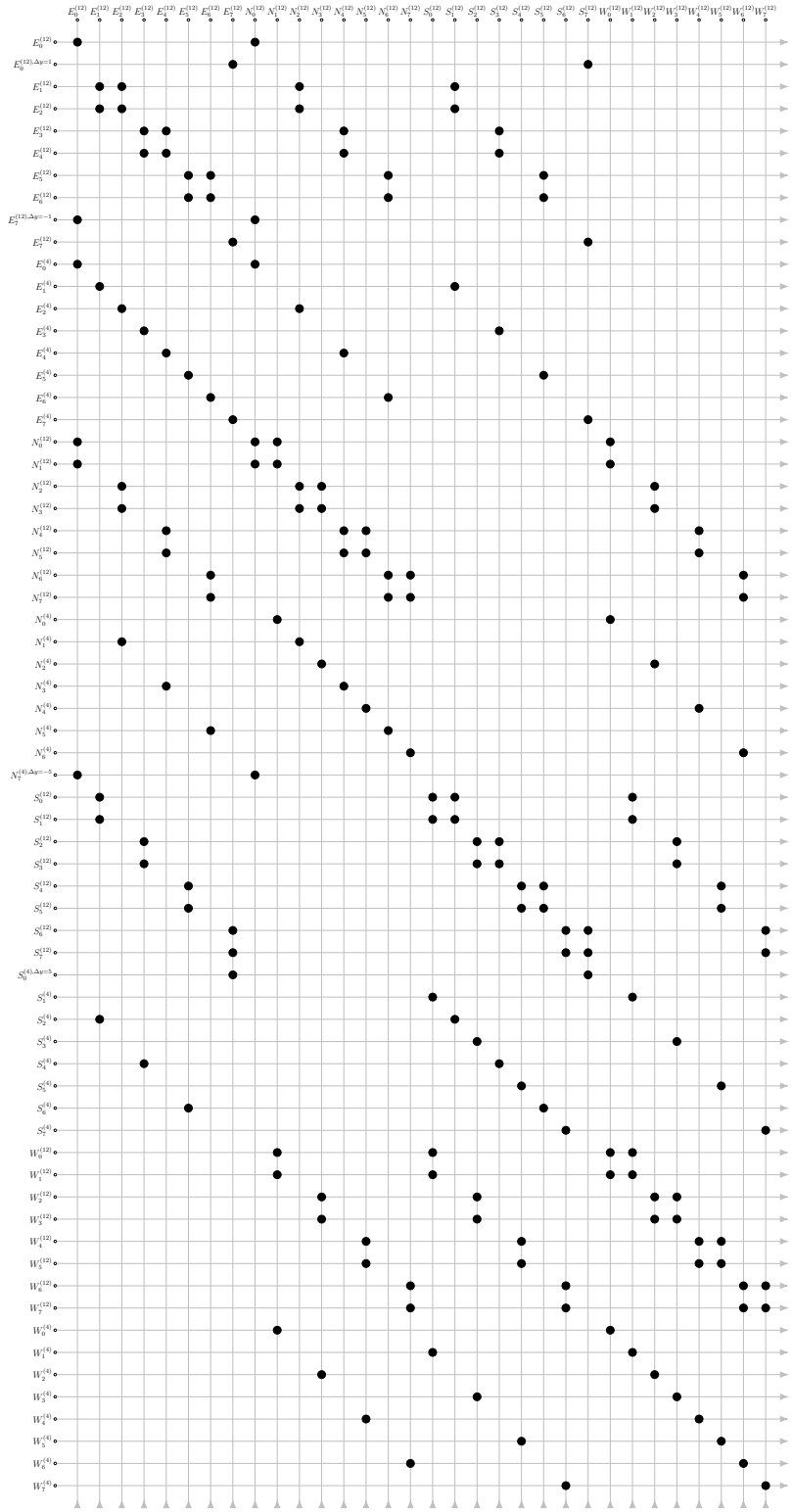


Figure 3.27: Connectivity between long wires in the UltraScale interconnect

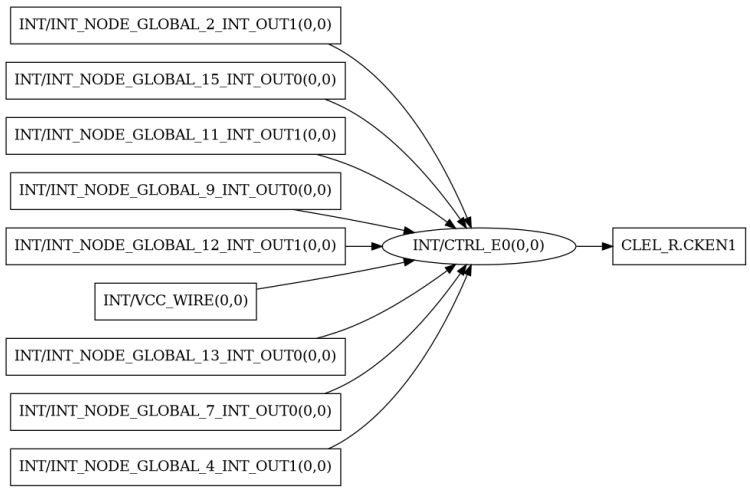


Figure 3.28: Clock-enable 1 input multiplexer

Chapter 4

Parametrised Cell Generators for FPGAs

BFG enables us to write parametrised cell generators, by writing C++ programs to emit custom circuitry and IC layout, for components of an FPGA fabric. In this chapter we present a catalogue of such generators and the principles of their design at the circuit, layout and software level. There are numerous ways to do this; we discuss three in Section 4.1. We cover basic elements such as flip-flops, buffers, tap cells, decap cells, and multiplexers in Section 4.3. We use these to assemble the first FPGA part, a 4-LUT-based Configurable Logic Block (CLB), in Section 4.4. We compare its performance to that of a standard-cell-based design with equivalent logic in order to demonstrate that our approach of generating cells from the bottom-up has merit over the top-down approach of eFPGA generators. In Sections 4.6, 4.7 and 4.8 we will show increasingly sophisticated generators for an S-44-based CLB, and $N:1$ and $M:2$ one-hot transmission-gate multiplexers. The latter of these derive from our work on reverse-engineering the UltraScale interconnect in Chapter 3. Finally, we synthesise the work on these cells, interconnect topology generation, and key results from literature to develop an example columnar architecture in Section 4.10. We attempt to capture this in a generator for a single, repeatable tile that could be used to build a whole fabric.

4.1 Paradigms

Software engineers understand that software design patterns, a concept borrowed from the architecture of physical buildings, help to speed up the development process by providing reusable, well-argued, field-tested solutions to common problems [10]. Design patterns simplify a programmer's, or system designer's, overall problem, leaving them to focus on the novel, interesting, or most important parts. To our knowledge, no such design patterns exist in the literature for the generation of IC layout. We imagine that the expertise in this domain is deeply intertwined with commercially-valuable trade secrets, if it isn't the secret itself.

On the other hand, there are many patterns for the generation of geometry and layout

in general. Computer-Aided Design (CAD) in many different fields requires automation of layout to some extent. Web developers must frequently express the layout of elements on a web page, as must mobile developers for a user-facing program on a smartphone. We can look to these domains for hints. What makes IC layout unique is the sheer number of precise constraints between nearby objects. Yet, even this scale is not unheard of: video game engines must often process complex relationships between many entities in real time.

In this work, we contribute examples of several patterns to use (or to not use) in programming IC layout. It is our hope that the community build on these ideas so that, in time, robust design patterns can emerge. We will discuss their pros and cons but do not claim that any are optimal. With that said, BFG's generators for FPGA cells are broadly generated in one of three ways.

The first layout design pattern is to simply **import** an existing layout and parametrise it (where licensing permits). This is how BFG's buffer, flip-flop, tap, decap, exclusive-or and active 2:1 multiplexer are built (Sec. 4.3). BFG includes an import tool that can turn any existing layout into the set of BFG C++ library calls that would reproduce it[1]. Since the layouts are static, imported designs come with hard-coded coordinates and no semantic information about the relationships between shapes. They nonetheless serve as an excellent starting point: after import, a designer can isolate sections and parameterise the code to depend on whatever they deem necessary. Each of the aforementioned cells will demonstrate an example of this. The fundamental limitation of this approach is that it's often unknown why a layout was made the way it was; some parameters are easy to change (such as changing the cell width or height), but others are not (transistor widths). This is because the interdependence of structures, such as transistors, is fixed. One is free to add some back in, but by that point one will have re-implemented the cell to be more like our second design pattern.

The second layout design pattern is to write complex, detailed, monolithic generators that are responsible for every component and each of their relationships. Wire paths are specified through parametric shapes (such as a `geometry::PolyLine`), but must be explicitly programmed to adapt to any of the structures that might affect it. One is still free to use software abstraction, but they will typically do so only in the context of the current cell. Reusable functions will be employed, for example, to create sets of transistors, repeating wire structures, and the like. The benefit of this approach is that a designer intent on a precisely optimised layout can achieve it; the detriment is that doing this is complicated, error-prone, and tedious. The `Sky130Mux` generator (Sec. 4.3.8) is our only example of this approach, because we refused to do it ever again after that.

The third layout design pattern is the most enlightened, where some control is sacrificed for additional abstraction. The best examples of this approach are the $N : 1$ and $M : 2$ one-hot transmission gate muxes in Sections 4.7 and 4.8. There is a temptation for a cell designer to calculate and draw a single rectangle for all of the PMOS diffusion regions in the

[1]The input must be VLSIR layout format, but that is attainable from common GDS format through another community tool, as discussed in Sec. 2.8.1.

transmission gate stack at their core. That might seem elegant at first glance, but we resist this temptation. Instead, we start by defining a transistor cell, the `Sky130SimpleTransistor`, two of which are joined to make a `Sky130TransmissionGate`. The transistors can be placed anywhere and come with multiple alignment points. Any number of transmission-gates are then joined in a `RowGuide` to form a `Sky130TransmissionGateStack`. More upfront effort is required to develop these sub-components, because each is significantly more complicated than if it had been created expressly for use in one of these cells. For example, the transistors can be configured to be PMOS or NMOS, with varying length and width, with variable padding on the left and right, and so on. The transmission-gate stack generator can take any number of inputs, external padding, height, and more. Only a limited subset of these parameters would have been needed for a specific instance. They are nevertheless more tractable to program because the total number of concerns is any single generator is much more limited than in the monolithic pattern. Lastly, commercial tools and other AMS frameworks also provide transistor abstractions, so this probably a good idea.

4.2 Usage

Together with a handful of extra tools, BFG can forms the core of an iterative end-to-end AMS design flow. As described in Section 2.8, VLSIR-based converter tools are required to translate BFG's VLSIR outputs into common LEF/DEF, GDSII and SPICE formats. Additional third-party tools are also required for DRC checking, LVS checking, SPICE simulation and viewing the final layouts. We primarily used FOSS offerings for each, namely:

1. Magic VLSI [72], for DRC checking SkyWater 130 nm;
2. KLayout [49], for viewing final layouts in GDSII format (and additional DRC checking); and
3. Xyce [115], a fully-featured SPICE simulator.

We also used commercial equivalents where necessary, such as in Section 4.4.3.

BFG is a command-line tool. The most convenient way to invoke it is to have it run a single generator as shown in the example below. The generator is specified by name, which we find helpful to extract into the shell variable `GENERATOR`. The generator's parameters are specified in a text-format Protocol Buffer at the path given by `--params`. The output VLSIR layout and circuit libraries are prefixed with whatever is given by the `--output_library` flag. Technology information is respectively specified by the `--technology`, `--primitives` and `--external_circuits` flags. Once BFG has completed successfully, `proto2gds` is called to convert the output layout into GDSII, which can then be opened in KLayout.

```
export GENERATOR=Sky130InterconnectMux2
./bfg \
    --jobs 0 \              # Use max. no. of threads for routing.
```

```
    --technology /path/to/sky130.technology.pb \
    --primitives /path/to/sky130.primitives.pb \
    --external_circuits ../sky130hd.pb  \
    --logtostderr \          # Log to console.
    --write_text_format \    # Write human-readable protos.
    --run_generator ${GENERATOR} \
    --params ${GENERATOR}.params.pb.txt \
    --output_library ${GENERATOR}

/path/to/proto2gds \
    --verbose \
    -i ${GENERATOR}.library.pb \
    -t /path/to/sky130.technology.pb \
    -o ${GENERATOR}.gds
```

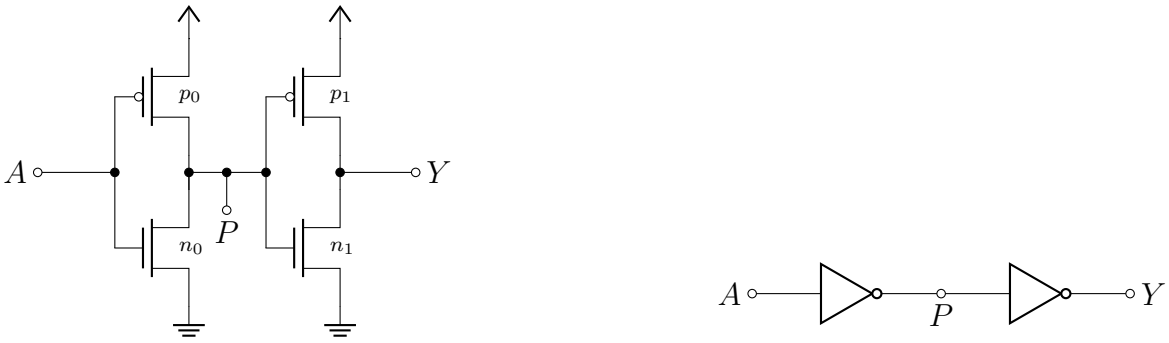
4.3 Basic Elements

4.3.1 Buffer

✓ DRC clean ✓ LVS pass

`atoms/sky130_buf.h`
`atoms/sky130_buf.cc`

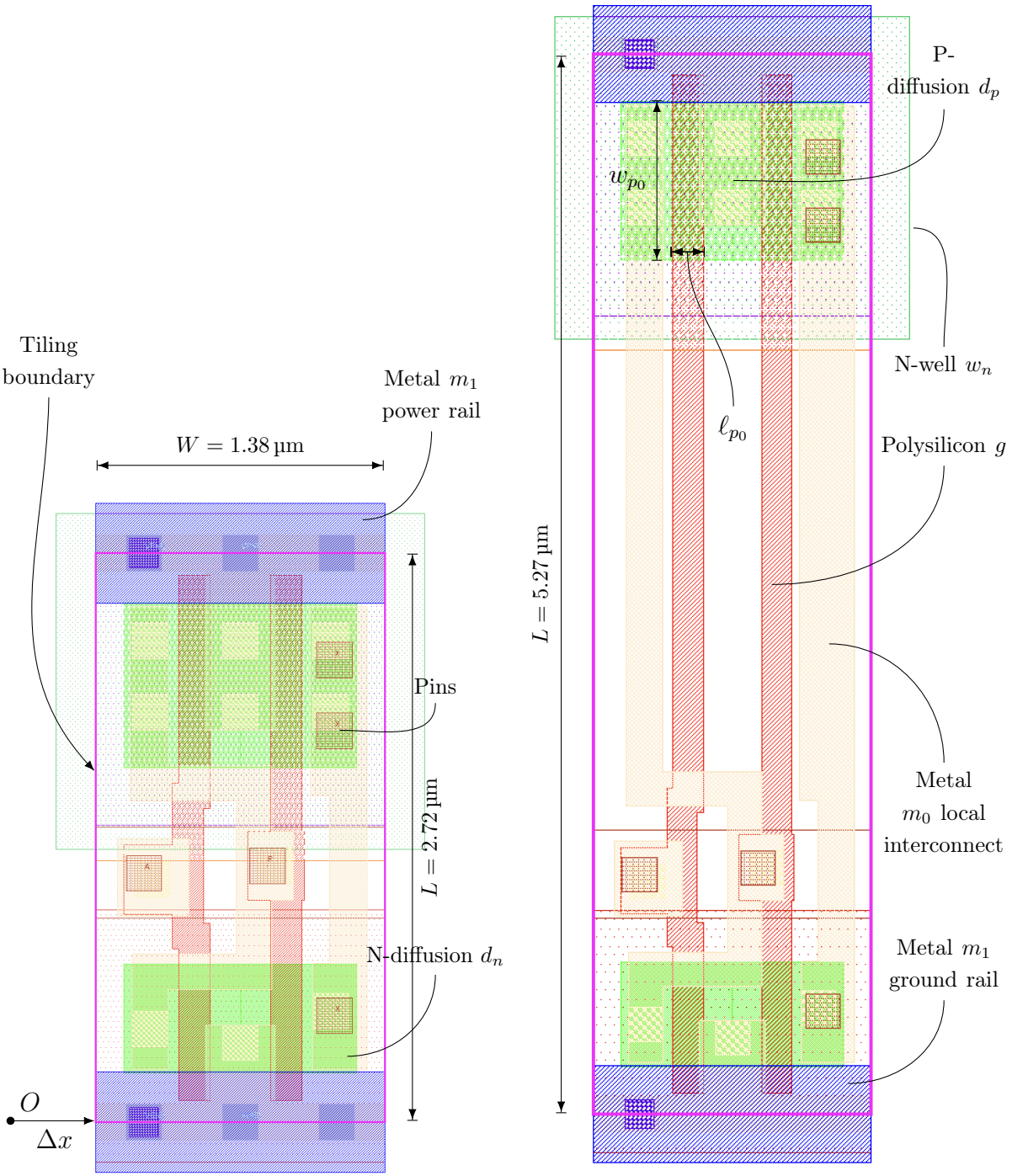
The `Sky130Buf` class is a straightforward generator for the buffer circuit of Figure 4.1a. We will later refer to this structure by the common logic gate notation shown in Figure 4.1b. The class was made by importing the SkyWater 130 nm `sky130_fd_sc_hd__buf_1` cell and modifying it to support the parameters listed in Table 4.1. The transistor widths and lengths can be set by the input, but their range is limited to what the existing structure could support without further modification. The output port P was also added; the SkyWater 130 nm library cell does not expose a port there.



(a) Shematic with transistors (b) Shorthand

Figure 4.1: Buffer circuit

The default parameters result in the layout of Figure 4.2a. Changing the cell height moves the PMOS structure up; if the input port A and output port P were also programmed to move, wider PMOS and NMOS transistors could also be accommodated. To accommodate off-pitch abutment with cells above and below, the vias connecting to the power and ground rails at the top and bottom of the cell can also disabled. Figure 4.2b shows some of these modifications.



(a) Default parameters

(b) Modified height; power and ground vias disabled

Figure 4.2: Generated buffer layouts

Table 4.1: Parameters exposed by the Sky130Buf simple buffer generator

Ref.	Parameter	Default
W	Cell width	1380 nm
H	Cell height	2720 nm
w_{p_0}, ℓ_{p_0}	p_0 Width, Length	790 nm, 150 nm
w_{p_1}, ℓ_{p_1}	p_1 Width, Length	790 nm, 150 nm
w_{n_0}, ℓ_{n_0}	n_0 Width, Length	520 nm, 150 nm
w_{n_1}, ℓ_{n_1}	n_1 Width, Length	520 nm, 150 nm
Δx	x-shift	0 nm
	Label pins	Yes
	Draw edge pins and vias	Yes

4.3.2 Split Buffer

✓ DRC clean ✓ LVS pass

`atoms/sky130_split_buffer.h`
`atoms/sky130_split_buffer.cc`

The `Sky130SplitBuffer` class generates a split buffer, so named because the buffered output X and inverted output P have independent paths to the input A. The schematic for the split buffer is shown in Figure 4.3 and its layout in Figure 4.4. Splitting the outputs allows the designer to reduce the delay between generation of a buffered signal and its complement, a feature that will become useful for the CLB in Section 4.4.

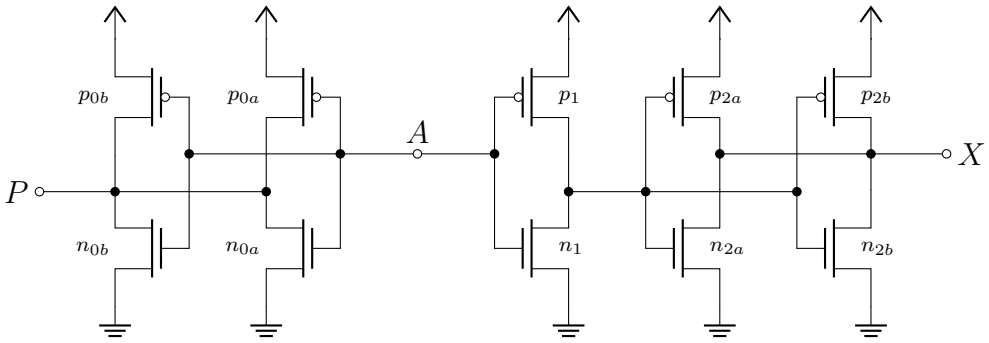


Figure 4.3: Split buffer circuit

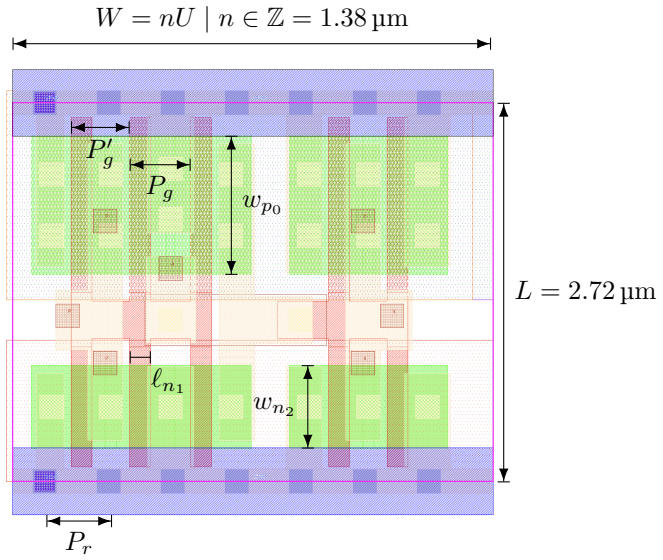


Figure 4.4: Layout of the Split Buffer

Unlike the `Sky130Buf`, the split buffer is built from scratch around the `Sky130SimpleTransistor` transistor generator (see Section 4.7.1). Its construction is straightforward. The layout is designed to stretch in most directions, being most restricted by the placement of polysilicon connections in the vertical centre of the cell; diffusion regions cannot come too close to polysilicon contacts. The N-well, `psdm` and `nsdm` regions can optionally be extended to the edges of the boundary to ensure abutment with neighbouring cells, at the designer's discretion.

The generator is designed so that the output stage can stack two or more transistors in parallel for additional drive strength, though only up to two transistors are currently implemented. By setting appropriate parameters, the b-subscript transistors in the schematic n_{0b}, p_{0b}, n_{2b}, and p_{2b} can be omitted. The layout is arranged so that additional transistors can be stacked to the left to strengthen the P output, and to the right to strengthen the X output. That is, the cell can easily grow left and right. In reality the output transistors must be balanced with to the input signal and the n_1, p_1 first-stage inverter, so the growth is somewhat bounded.

Supported parameters are listed in Table 4.2. In addition to transistor sizing, the designer can independently tweak the polysilicon spacing between stacked and non-stacked gates. Additionally, unlike some other cells, a "unit width" can be specified to force the width of the cell into the minimum viable integer multiple of the given value. By default, the value from SkyWater 130 nm standard library cell library (460 nm) is used.

Table 4.2: Parameters exposed by the Sky130SplitBuffer split buffer generator

Ref.	Parameter	Default
U	Unit width	
H	Cell height	2720 nm
w_{p_0}, ℓ_{p_0}	p_0 Width, Length	1000 nm, 150 nm
w_{p_1}, ℓ_{p_1}	p_1 Width, Length	1000 nm, 150 nm
w_{p_2}, ℓ_{p_2}	p_2 Width, Length	1000 nm, 150 nm
w_{n_0}, ℓ_{n_0}	n_0 Width, Length	600 nm, 150 nm
w_{n_1}, ℓ_{n_1}	n_1 Width, Length	600 nm, 150 nm
w_{n_2}, ℓ_{n_2}	n_2 Width, Length	600 nm, 150 nm
	Double n_0 and p_0	Yes
	Double n_2 and p_2	Yes
P'_g	Poly pitch for adjacent transistors	420 nm
P_g	Poly pitch otherwise	440 nm
	Draw edge pins and vias	Yes
P_r	Rail via pitch	460 nm
	Draw wells to boundary	Yes

4.3.3 Flip-flop

✓ DRC clean ✓ LVS pass

`atoms/sky130_dfxtp.h`
`atoms/sky130_dfxtp.cc`

BFG's `Sky130Dfxtp` class implements a delay flip-flop generator based on the SkyWater 130 nm `sky130_fd_sc_hd__dfxtp_1` cell (hereafter the base cell), which was imported and parameterised. The first modification made to the base cell was to allow the designer to disable the input clock buffer. Its schematic and layout, without an input clock buffer, are shown in Figures 4.5 and 4.6a respectively. In cases where several flip-flop cells are placed next to each other they can share a single external clock buffer instead, thereby saving area. This feature is used extensively by the CLB (Sec. 4.4) and interconnect multiplexers (Sec. 4.7, 4.8). The input clock buffer is almost identical to the buffer generated by the `Sky130Buf` class. When enabled, the layout looks like Figure 4.6b.

The transistors in the output buffer $n_i, p_i \mid i \in \{8, 9\}$ can be also be resized, within the limits of a four-transistor buffer like in `Sky130Buf` (Secs. 4.3.1). That is, there is no support for adding additional transistors in parallel at the output. The flip-flop is composed of two back-to-back latches which are timing sensitive, and the first inverter in the output buffer participates intimately in the latch-to-latch feedback loop. As a result, the sizes of transistors in the core latch complex are not exposed, and transistors n_8 and n_9 should be modified with caution. Modifying the internal transistors does not achieve much for the designer since the flip-flop is internally consistent, and additional drive strength can be achieved through an additional output inverter.

It may be surprising, therefore, that the generator also adds an optional inverted output $\overline{Q}$. The load placed on $\overline{Q}$ is likely to interfere with the flip-flop's operation, and should be tuned carefully. A summary of all available parameters is shown in Table 4.3.

Table 4.3: Parameters exposed by the Sky130Dfxtp flip-flop generator

Ref.	Parameter	Default
w_{p_8}, ℓ_{p_8}	p_8 Width, Length	1000 nm, 150 nm
w_{p_9}, ℓ_{p_9}	p_9 Width, Length	1000 nm, 150 nm
w_{n_8}, ℓ_{n_8}	n_8 Width, Length	650 nm, 150 nm
w_{n_9}, ℓ_{n_9}	n_9 Width, Length	650 nm, 150 nm
	Draw input clock buffer	No
	Add inverted output port, $\overline{Q}$	Yes

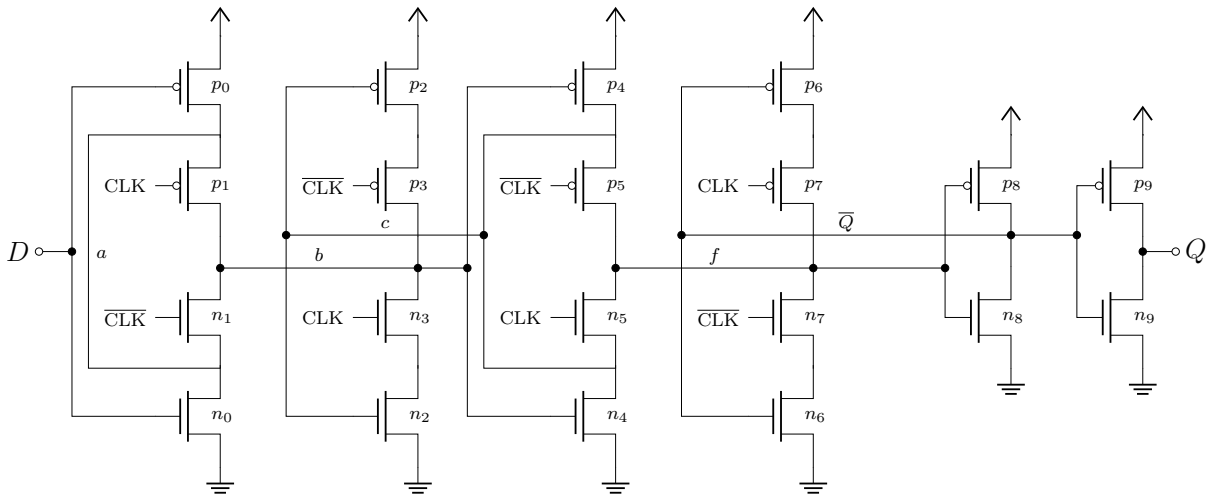
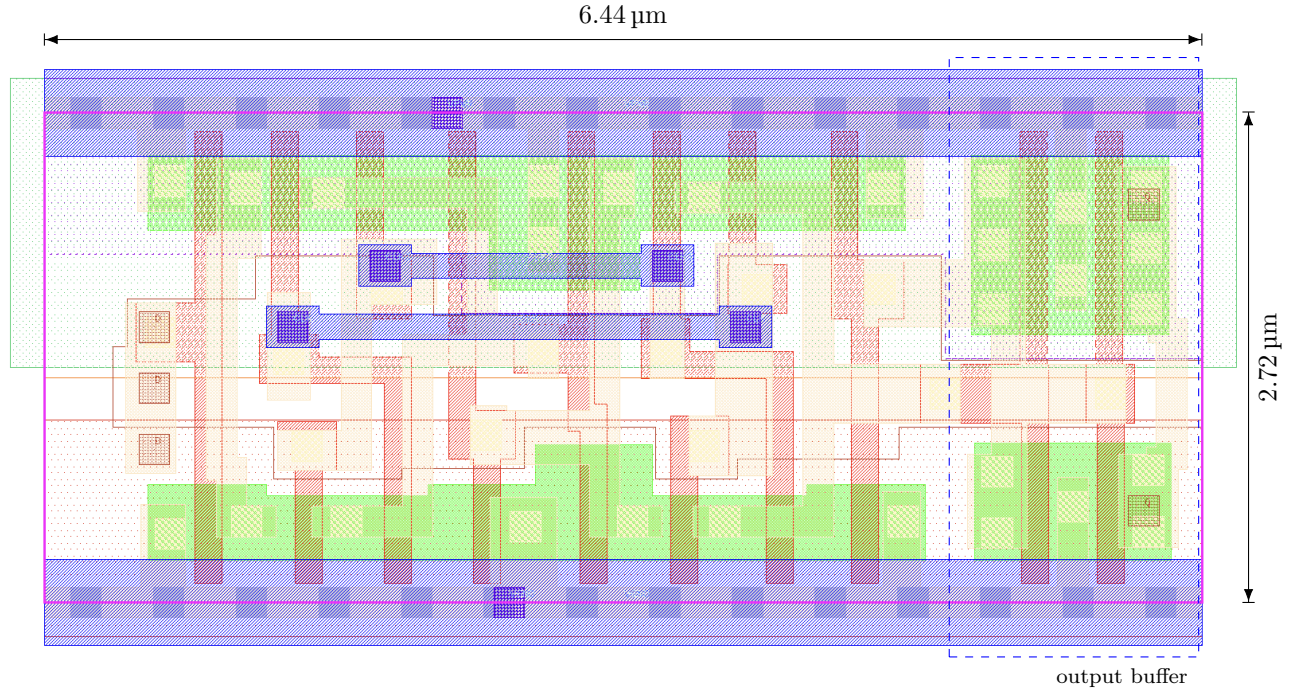
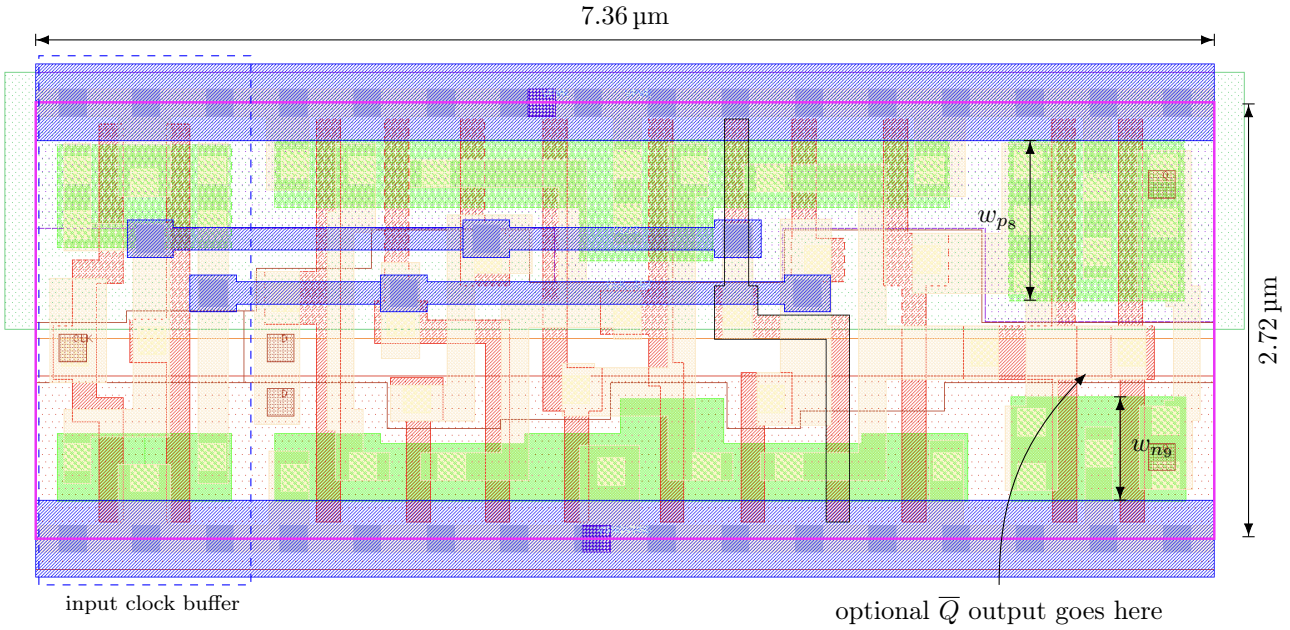


Figure 4.5: A flip-flop circuit, without the input clock buffer



(a) Without input clock buffer



(b) With input clock buffer

Figure 4.6: Generated flip-flop layouts

4.3.4 Tap

✓ DRC clean ✓ LVS pass

`atoms/sky130_tap.h`
`atoms/sky130_tap.cc`

"Tap" cells provide connections to the substrate in each well. These connections can either be used to bias transistor threshold voltages V_T or to clamp substrate potentials so that they do not drift. In order to prevent the phenomenon of latch-up [27], tap connections must be made at regular intervals in the transistor plane so that no transistor is too far from a tap.

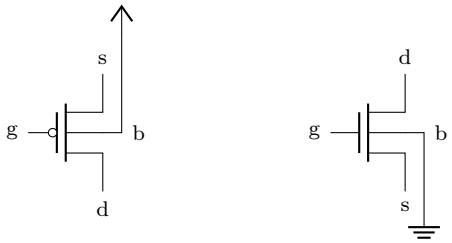


Figure 4.7: Default MOSFET bulk connections by nearby tap cells

Accordingly, BFG's cells regularly connect the substrates of P-type MOSFETs to the power rail and the substrates of N-type MOSFETs to the ground rail. With BFG it is practical to add substrate connections directly into each cell. But because we mix free-form layout with the standard-cell methodology of fixed-height rows, we borrow the standard-cell tap insertion doctrine too: using the `RowGuide` structure (Section 2.3.4), tap cells are automatically inserted at foundry-mandated intervals as new cells are added. Throughout this chapter, PMOS and NMOS FETs will be drawn without their bulk (substrate) connections. Through the correct placement of nearby tap cells, these should be considered to be connected to power and ground, respectively, as shown in Figure 4.7.

The tap cell itself is simple (Fig. 4.8). In the south, a cutout is made for connection to the substrate, and this cutout is then connected to the horizontal ground rail on `met1.drawing` through the `li1.drawing` layer and appropriate vias on `licon.drawing` and `via.drawing`. This is repeated for connection to the power rail in the north, but within an N-well (a `nwell.drawing` box). Table 4.4 lists the available parameters for the tap cell generator, `Sky130Tap`, in `src/atoms/sky130_tap.h` of the BFG source repository.

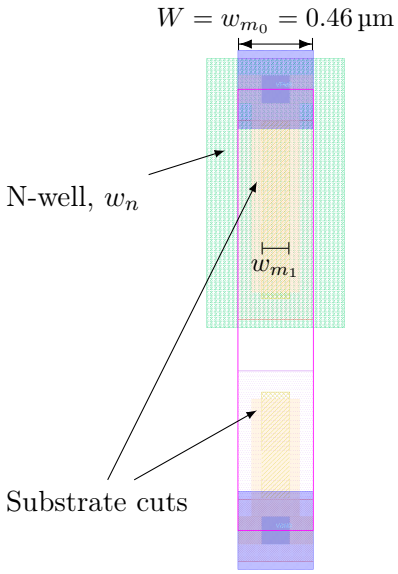


Figure 4.8: Tap cell layout

Ref.	Parameter	Default
W	Cell width	460 nm
H	Cell height	2720 nm
w_{m_0}	Metal m_0 width	170 nm
w_{m_1}	Metal m_1 width	460 nm

Table 4.4: Parameters exposed by the `Sky130Tap` tap cell generator

4.3.5 Decoupling Capacitor

`atoms/sky130_decap.h`
`atoms/sky130_decap.cc`

Decoupling capacitor (decap) cells are used to fill otherwise-unused space in the layout. They connect the P- and NMOS transistors in such as way that their gate capacitance can be used to store charge; see Figure 4.9. The decap generator parameters are summarised in Table 4.5 and some are annotated on Figure 4.10c.

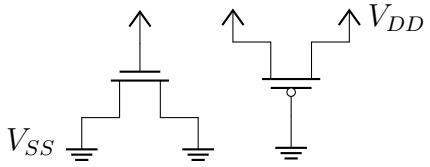


Figure 4.9: Decap cell circuit

The decap cell is based on the SkyWater 130 nm `sky130_fd_sc_hd__decap_1` cell but does not import the design. Rather, the design is simple enough to recreate from scratch. The PMOS and NMOS gates are sized to maximise their gate area within the cell — note that the parameters list these as automatic, not as static defaults. This generator provides the simplest complete example of how geometric reasoning can be encoded to develop layout generators. The pertinent constraints, which are labelled on Figure 4.10b, are:

A. The minimum separation between metal m_0 shapes in this cell and any neighbours. By convention, we assume that shapes in neighbouring layouts are equidistant from the vertical edges. A similar constraint at this edge also defines the minimum separation of diffusion regions d_n, d_p from the edge.

B. A minimum encapsulation rule defines how far in from the m_0 and diffusion boundaries a via v_0 can be placed. These vias connect the source and drain regions of the transistor to m_0, which in the PMOS case is connected to V_{DD} and in the NMOS case is connected to ground.

C. A similar constraint defines the minimum distance to the polysilicon pour on the other side of the via. Together, *A*, *B*, the width of vias on v_0 and the via-to-polysilicon distance define how far from the edge polysilicon can start, which defines the maximum length ℓ of the polysilicon gate for a given cell width W:

$$\ell = W - 2C$$

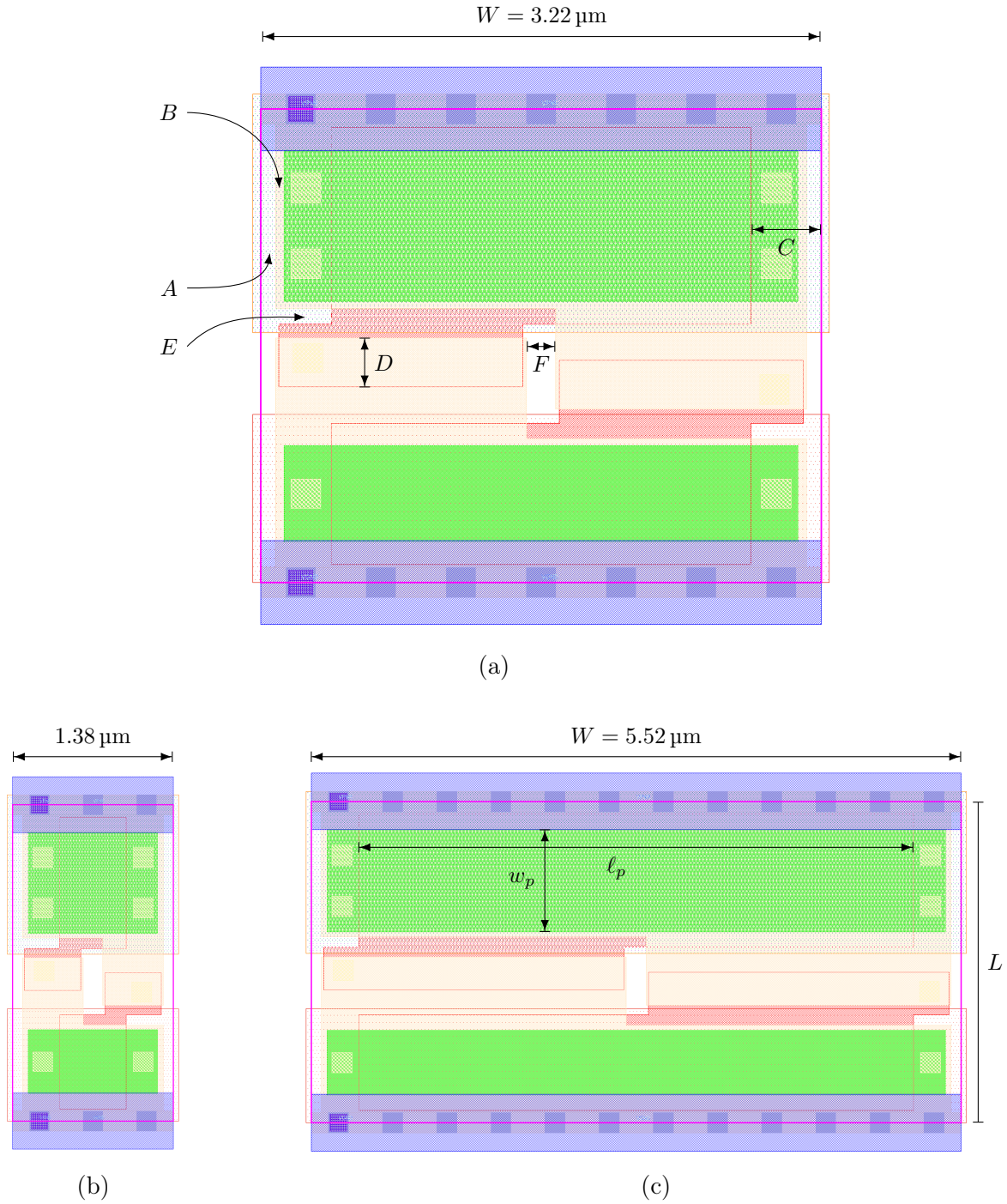


Figure 4.10: Generated decap cell layouts in three sizes

D. Connection to the opposite rail (ground for PMOS, V_{DD} for NMOS) must be made

by connecting the gate of the transistor to the corresponding m_0 from the opposite side of the cell. Polysilicon g contacts (also the v_0 layer) must be placed at least some distance inside from the edge of the polysilicon, and we must also observe the minimum via height. A similar set of rules apply for v_0 as encapsulated by m_0 above it. Together, D defines the minimum overlap of metal and polysilicon for the contact.

E. Polysilicon contacts must occur at a minimum distance from the gate (where the polysilicon overlaps diffusion), which defines the closest we can put those contacts to either transistor. Because we must adhere to these rules both from the PMOS side of the gate to the NMOS m_0 contact, and from the NMOS side to the PMOS m_0, we have defined the minimum separation between the two transistors. The minimum distance polysilicon can come to the north and south edges is defined (as with A), as half the minimum separation to another polysilicon shape. Together, these values constrain the total width of the transistors for a given cell height; we split the total width between the PMOS and NMOS roughly according to the compensatory ratio for P- versus N-type charge mobility (typically $1.6:1$ in SkyWater 130 nm).

F. Finally, analogous constraints in the transverse direction dictate the *minimum* width of the cell. Consider Figure 4.10b, which shows the narrowest the cell can be without any rule violations. If the cell were to be made narrower, either the m_0 to m_0 minimum separation, the polysilicon minimum width, or at least one of the earlier rules would be violated.

Table 4.5: Parameters exposed by the `Sky130Decap` decap cell generator

Ref.	Parameter	Default
W	Cell width	1380 nm
H	Cell height	2720 nm
w_0, ℓ_0	p_0 Width, Length	Automatic
w_1, ℓ_1	n_1 Width, Length	Automatic
	Label pins	Yes
	Draw edge pins and vias	Yes
P_r	Rail via pitch	460 nm

Once we have the flexibility to generate almost-any size of decap cell, it becomes very simple to offer a facility for using them to fill almost-any available space. This is usually considered a separate step in the digital VLSI toolchain, but here it is a built in function. Listing 4.1 shows how simple it is to generate the minimum number of decap cells to fill a span in some row, as long as we have a span greater than the minimum decap width:.

Listing 4.1: Method to fill decaps into some span of a row, to the right, from `atoms/sky130_decap.cc`

```
556 void Sky130Decap::FillDecapsRight(const std::optional<Parameters> &base_params,
557                                   int64_t span,
558                                   RowGuide *row) {
559   if (span < Parameters::kMinWidthNm) {
560     return;
561   }
562   const PhysicalPropertiesDatabase &db = row->design_db()->physical_db();
563   std::vector<int64_t> decap_widths = Utility::StripInUnits(
564       span,
565       Parameters::kMaxWidthNm,
566       Sky130Parameters::kStandardCellUnitWidthNm,
567       Parameters::kMinWidthNm);
568   int num_decaps = 0;
569   if (row->empty()) {
570     // Maybe we need to be able to specify a default height.
571     return;
572   }
573   uint64_t height_nm = row->Height();
574   for (int64_t width : decap_widths) {
575     Cell *decap_cell = Make(base_params,
576                             db.ToExternalUnits(row->Height()),
577                             db.ToExternalUnits(width),
578                             row->design_db());
579     row->InstantiateBack(absl::StrCat(decap_cell->name(), "_i", num_decaps),
580                          decap_cell);
581     ++num_decaps;
582   }
583 }
```

The core of this method is the utility method `StripIntoUnits`, which we define as follows. Given a unit width u, maximum width M, an optional minimum width m and some span to cover s, we split S into the minimum number of sub-lengths s_i such that:

- s_i are integer multiples of u

$$s_i = nu \mid n \in \mathbb{Z}$$

- no segment exceeds the maximum and all at least equal the minimum

$$m \leq s_i \leq M \quad \forall i$$

- all of the segments sum to within one unit of, but do not exceed, the span

$$\hat{S} \triangleq \sum_i s_i$$

$$\hat{S} < S$$

$$S - \hat{S} < u$$

The method is one of many commonly needed utilities in BFG's geometric routines, shown in Listing 4.2. See `utility.{h,cc}` for more.

Listing 4.2: The `StripIntoUnits` utility method, from `utlity.cc`

```
121  // Does a kind of apriori stripmining, splitting 'length' into as many
122  // sub-lengths of 'max' size as can fit, and any remaining length, where all
123  // returned length are multiples of the given 'unit'. (The 'max' value is also
124  // rounded down to the nearest multiple of 'unit'.)
125  //
126  // If 'min' is given, some attempt is made to rebalance the distribution of
127  // units so that no quantity is less than the min. Quantisation in multiples
128  // of unit will still occur.
129  //
130  // e.g. StripInUnits(32, 9, 3) will result in [9, 9, 9, 3], with 2 leftover
131  // of the original quantity not fitting a multiple of 3.
132  // StripInUnits(33, 9, 3) will result in [9, 9, 9, 6].
133  // StripInUnits(32, 8, 3) will result in [6, 6, 6, 6, 6].
134  static std::vector<int64_t> StripInUnits(
135      int64_t length, int64_t max, int64_t unit,
136      std::optional<int64_t> min = std::nullopt) {
137    // Rely on truncating (floor) behaviour.
138    int64_t real_max = (max / unit) * unit;
139
140    std::vector<int64_t> lengths;
141
142    // Stripmining!
143    int64_t unallocated = length;
144    while (unallocated >= unit) {
145      int64_t remainder = unallocated - real_max;
146      if (remainder >= 0) {
147        lengths.push_back(real_max);
148      } else {
149        // Again we rely on truncating (floor) behaviour:
150        lengths.push_back((unallocated / unit) * unit);
151      }
152      unallocated = remainder;
153    }
154
155    if (lengths.empty()) {
156      return lengths;
157    }
158
159    // The last length is the smallest, by construction. But we might need to
160    // borrow from several previous lengths to get it up to the minimum size.
161    if (min && lengths.back() < *min) {
162      DCHECK(*min <= max);
163
164      int64_t real_min = unit * ((*min) / unit);
165      int64_t overflow = real_min - lengths.back();
166
167      for (auto it = lengths.rbegin() + 1; it < lengths.rend(); ++it) {
168        int64_t borrowable = (*it) - real_min;
169        if (borrowable <= overflow) {
170          (*it) -= borrowable;
171          lengths.back() += borrowable;
172          overflow -= borrowable;
173        } else {
174          (*it) -= overflow;
175          lengths.back() += overflow;
176          break;
177        }
178      }
179      if (overflow > 0) {
```

```
180         // Could not satisfy the minimum.
181         LOG(ERROR) << "Could not satisfy requirement for minimum value";
182       }
183     }
184
185     return lengths;
186   }
```

4.3.6 Active 2 : 1 Multiplexer

✓ DRC clean ✓ LVS pass

`atoms/sky130_hd_mux2_1.h`
`atoms/sky130_hd_mux2_1.cc`

The `Sky130HdMux21` generator captures a simple 2 : 1 active multiplexer. It is so named because it is reproduces the SkyWater 130 nm `sky130_fd_sc_hd__mux2_1` cell almost verbatim; we call it *active* because each internal logic node is always pulled to either the power or ground rail. Although this is usual in CMOS logic design, it stands in contrast to the *passive* multiplexers we will see in Sections 4.3.8, 4.7 and 4.8.

Unfortunately, this cell is not particularly interesting, and makes few modifications to the original library version by default. The generated circuit and layout are shown in Figure 4.11 and 4.12 respectively. The most interesting parameters to tweak *would* be the output buffer transistors, but even these have not yet been necessary and are left only partially implemented[2]. The cell is nonetheless a crucial component of the `Sky130Carry1` carry-chain bit in Section 4.5. Since it is so uninteresting, we will amuse the curious reader by deriving its logical function algebraically. Start with the conditions necessary to pull the node $\overline{X}$ high,

$$\overline{X} = \overline{A_1} \cdot \overline{\overline{S}} \left(A_0 + \overline{\overline{S}} \right) + \overline{A_0} \cdot \overline{S} \left(A_1 + \overline{S} \right)$$

which we simplify to:

$$\begin{aligned} \overline{X} &= \overline{A_1} \cdot S \left(A_0 + S \right) + \overline{A_0} \cdot \overline{S} \left(A_1 + \overline{S} \right) \\ &= \overline{A_1} \cdot S \left(A_0 + 1 \right) + \overline{A_0} \cdot \overline{S} \left(A_1 + 1 \right) \\ &= \overline{A_1} \cdot S + \overline{A_0} \cdot \overline{S} \end{aligned}$$

The output inverter restores X by inverting the intermediate node,

$$\begin{aligned} X &= \overline{\overline{X}} \\ &= \overline{\overline{A_1} \cdot S + \overline{A_0} \cdot \overline{S}} \\ &= \overline{\left(\overline{A_1} \cdot S \right)} \cdot \overline{\left(\overline{A_0} \cdot \overline{S} \right)} \\ &= \left(A_1 + \overline{S} \right) \cdot \left(A_0 + S \right) \end{aligned}$$

[2]The parameters are used to set the model parameters in the *circuit* but not in the layout. Changing these from the default breaks the correspondence between layout and circuit. We leave completion of the necessary layout modifications to future work, or any interested reader.

We introduce the tautology $(S + \overline{S})$ and simplify again:

$$\begin{aligned}
&= A_1A_0 + \overline{S}A_0 + SA_1 + S\overline{S} \\
&= \left(S + \overline{S}\right) A_1A_0 + \overline{S}A_0 + SA_1 \\
&= SA_1\left(\overline{A_0} + 1\right) + \overline{S}A_0\left(\overline{A_1} + 1\right) \\
&= SA_1 + \overline{S}A_0
\end{aligned}$$

The mux selects the value of the A_0 or A_1 input if $S = 0$ or $S = 1$, respectively.

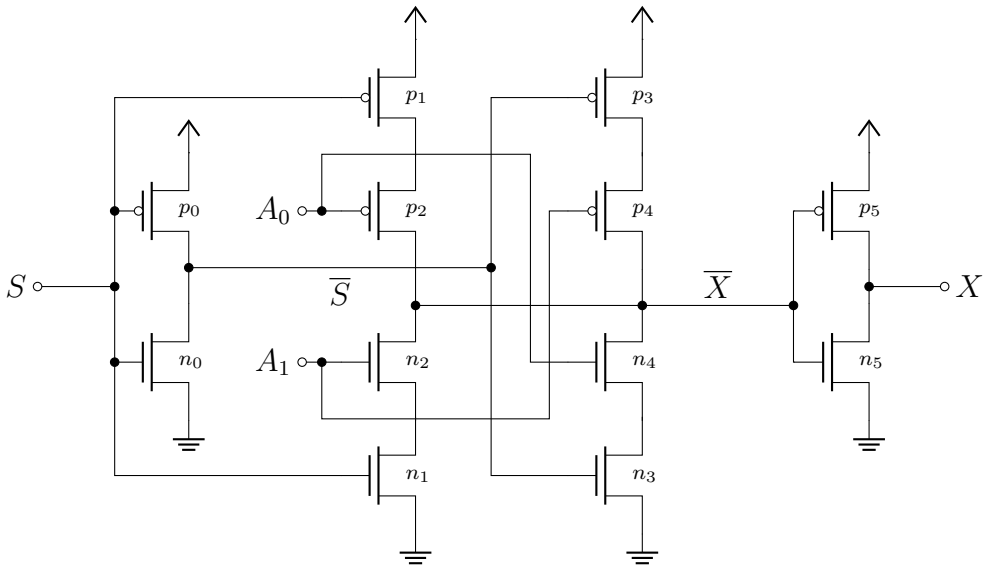
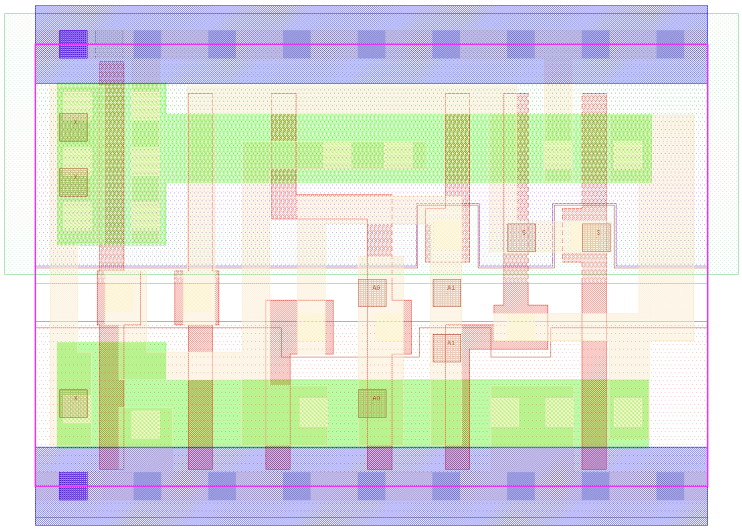


Figure 4.11: Active 2 : 1 multiplexer circuit



;

Figure 4.12: Active 2 : 1 multiplexer layout

4.3.7 Exclusive-Or Gate

✓ DRC clean ✓ LVS pass

`atoms/sky130_xor2.h`
`atoms/sky130_xor2.cc`

Like the active 2:1 multiplexer generator, the `Sky130Xor2` class generates a cell based on an imported SkyWater 130 nm layout, though in this case the source cell is `sky130_fd_sc_hd__xor2_1` and some additional parameters have been instrumented. The generated circuit, layout, and available parameters are shown in Figure 4.13, Figure 4.14 and Table 4.6, respectively. Although the generator exposes cell width and height as parameters, the internal structures have not been made to adapt very intelligently. The most useful parameters, as we will see in Section 4.5, are the pin labelling and rail insertion toggles.

We can verify the function of the circuit as follows. The output X is pulled low when

$$\overline{X} = AB + \overline{A + B}$$

thus by inversion the output is

$$\begin{aligned} X &= \overline{AB + \overline{A + B}} \\ &= \overline{AB} \cdot (A + B) \\ &= \left(\overline{A} + \overline{B}\right)(A + B) \\ &= \overline{A}A + \overline{A}B + \overline{B}A + \overline{B}B \\ &= \overline{A}B + \overline{B}A \\ &= A \oplus B \end{aligned}$$

Table 4.6: Parameters exposed by the `Sky130Xor2` two-input exclusive-or generator

Parameter	Default
Cell width	3220 nm
Cell height	2720 nm
Label pins	Yes
Draw edge pins and vias	Yes

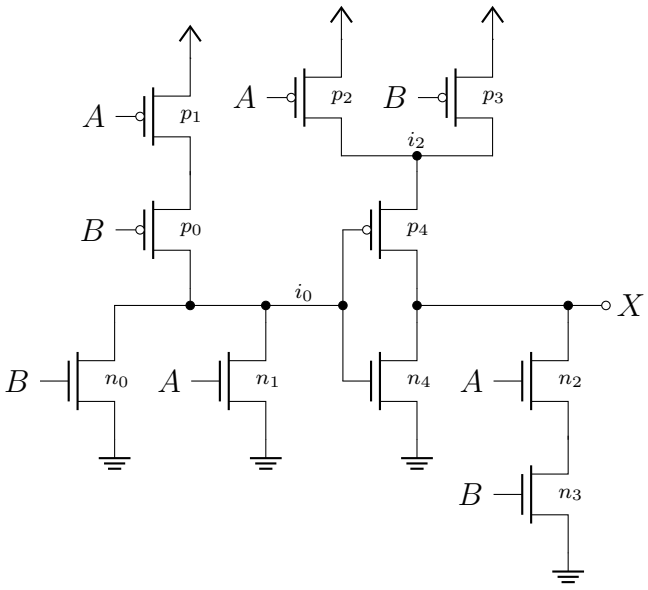


Figure 4.13: The `sky130_fd_sc_hd__xor2_1` circuit.

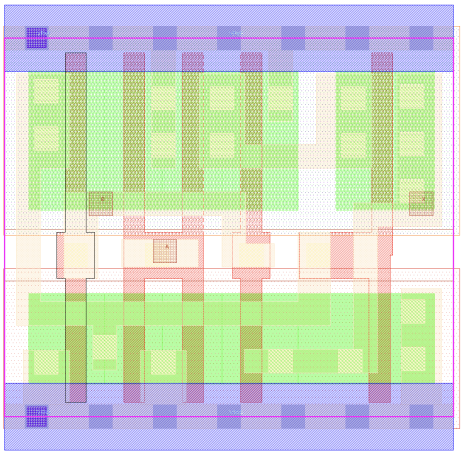


Figure 4.14: Layout generated by `Sky130Xor2`

4.3.8 Hierarchical $8:1$ Transmission-Gate Multiplexer

✓ DRC clean ✓ LVS pass

`atoms/sky130_sky130_mux.h`
`atoms/sky130_sky130_mux.cc`

Class `Sky130Mux` generates a hierarchical 8:1 transmission-gate multiplexer. For clarity, one can think of the circuit as implementing the mux tree in Figure 4.19, although in reality it implements the circuit of Figure 4.20. In the latter circuit the P- and NMOS trees are completely separated: there are no ties between the source/drain terminals of P- and NMOS devices at the intermediate nodes. It is more accurately described as a PMOS and an NMOS 8:1 pass-gate multiplexer tree tied together at the start and end. Avoiding the intermediate connections saves considerable wire area and, as a result, the design is reasonably compact.

This cell uses a monolithic approach to layout generation, as discussed in Section 4.1. There *are* some attempts at functional decomposition and reuse, but all of the layout features for the cell are contained within the class. Concerns are not well separated and, as we shall see, there are many interrelated components that a designer must keep track of in order to correctly construct the output. The code is, unsurprisingly, largely unreadable[3]. On the plus side, here is an example that demonstrates what it might take to get a complex cell right.

The final generated layout is shown in Figure 4.15. There are twenty-eight transistors, four of which form the final output $2:1$ mux in the centre of the structure. The overall generator must support sizing each of these transistors independently. If we remove the final-stage transistors, the remaining twenty-four form four groups of six transistors that are laid out in the same pattern. The final output is generated by instantiating the base pattern in the upper-left, then mirroring it horizontally in the upper-right, vertically in the lower-left, and in both directions in the lower-right (Fig. 4.16). The final output mux is placed in the centre at the end. The base pattern is generated by a single function; in order to support every use necessary we equip it with the following parameters:

- the width w_{f_i} and length ℓ_{f_i} of each transistor $f_i \mid i \in 0, \ldots, 5$;
- the diffusion layer to use (since some are N- and some are PMOS);
- any additional extension of the polysilicon gates to the north or south, per column;
- how far the input ports should be extended out from the transistors in x, and their separation in y;
- whether the input ports should be added at all; and
- any necessary y offset between (f_4, f_5) and (f_0, f_1, f_2, f_3).

[3]Though I must insist that you *do* read it: `atoms/sky130_mux.cc`.

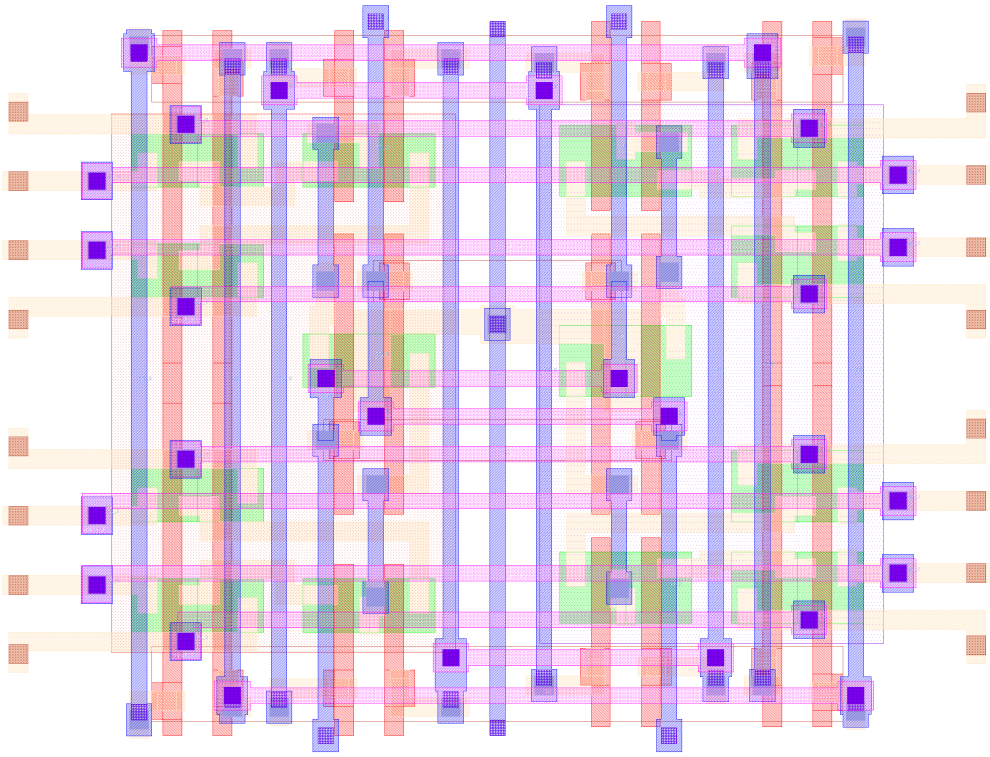


Figure 4.15: Generated layout of the `Sky130Mux`

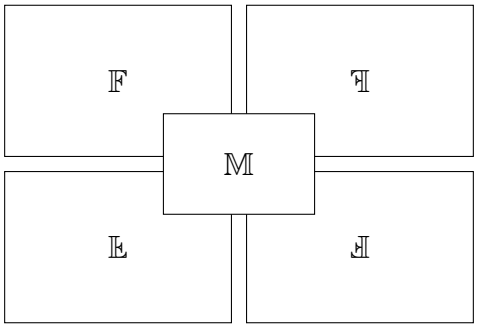


Figure 4.16: Composition of the final multiplexer from the base pattern $\mathbb{F}$ and output multiplexer $\mathbb{M}$

The complexity in this approach arises from keeping track of all the sets of constraints that apply to a single span or spacing. Figure 4.17 shows some of the rules for setting out just the base pattern between the diffusion d (in green) and polysilicon g (in red) layers:

1. Within a transistor we know that the diffusion must extend past the polysilicon by a minimum amount A (counting the spand under an adjacent gate).

2. Each source and diffusion region must also accommodate a via on v_0 to metal m_0. To fit a via, the diffusion must be wide and tall enough to accommodate the square dimensions of the via itself and the minimum spacing from the via to the polysilicon and diffusion edges.

3. If two horizontally adjacent transistors are not identically wide (i.e. $w_{f_0} \neq w_{f_2}$ as in in Fig. 4.15), we must also contend with the minimum separation required between the opposite polysilicon and the larger diffusion Z.

4. If the lengths of two vertically adjacent transistors do not match, a parallel edge is created by opposite the diffusion of the shorter gate (f_3 in Fig. 4.15) that must be set at at a minimum separation D from the polysilicon. The point at which the polysilicon narrows can be moved as long as the minimum extension of polysilicon past a gate (R) is not exceeded on the *other* side.

5. The more restrictive of (2) and the minimum polysilicon spacing D yields the actual spacing between the polysilicon gates S. The width of the polysilicon itself is given by the *length* of each transistor which, together with S, sets the gate pitch.

6. To minimise interactions with shapes outside of this structure, the top-row diffusion regions grow down and the bottom-row diffusions grow up. That way the horizontal edge at top and bottom is straight. The spacing between the rows is dictated by the the maximum of (3) and the minimum diffusion-to-diffusion spacing P.

7. If specified, the f_4 and f_5 transistors can be shifted in y (up and down).

8. The minimum horizontal offset to the f_4 and f_5 transistors is determined by the minimum diffusion separation P.

9. Together, the overall height of of the polysilicon Q, and hence the pattern, is set by the most restrictive combination of (4), (6) and (7); the overall width is set by (1), (2), (3), (5), and (8).

The pattern must also accommodate the metal m_0 wires that connect the gates together, shown in an excerpt from the final design in 4.18. (The polysilicon and metal m_0 wiring in the bottom right of the image are from the final $2:1$ multiplexer.) These wires have minimum width w_{m_0} and minimum separation G. They must properly encapsulate each via at each connection to the underlying diffusion and polysilicon. We must augment each of the rules numbered above with the constraints of the wiring layer and its connections. For example, the height of the cell must also be able to accomodate to horizontal wires E and F, vias M and N, and the wiring needed to connect to those vias and snake out J and K

Once the base patterns have been instantiated, the final 2:1 multiplexer is placed in the centre and all of the internal wiring must be laid. Each of these structures introduces constraints: the spacing within and between the base patterns must accommodate the final

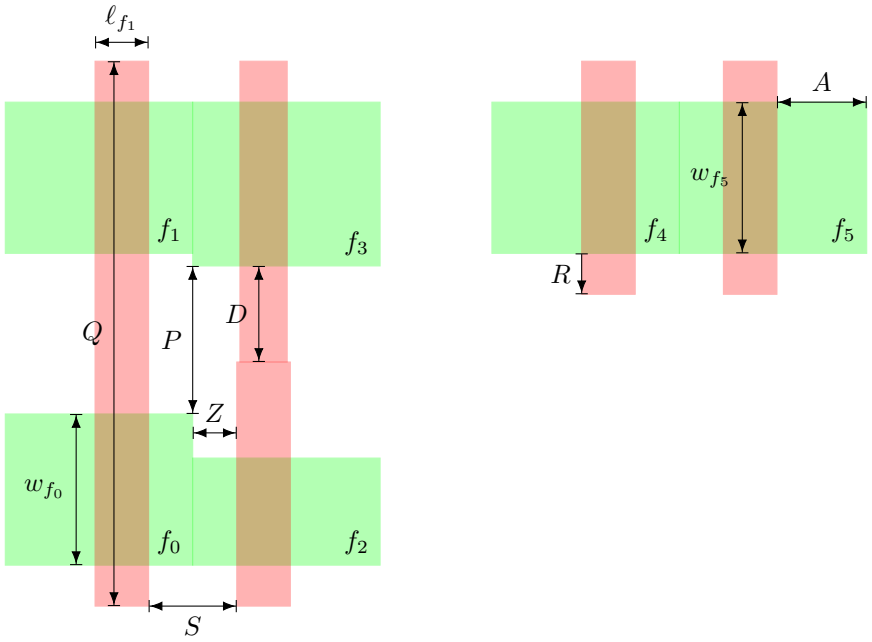


Figure 4.17: Base layout from which the `Sky130Mux` is composed

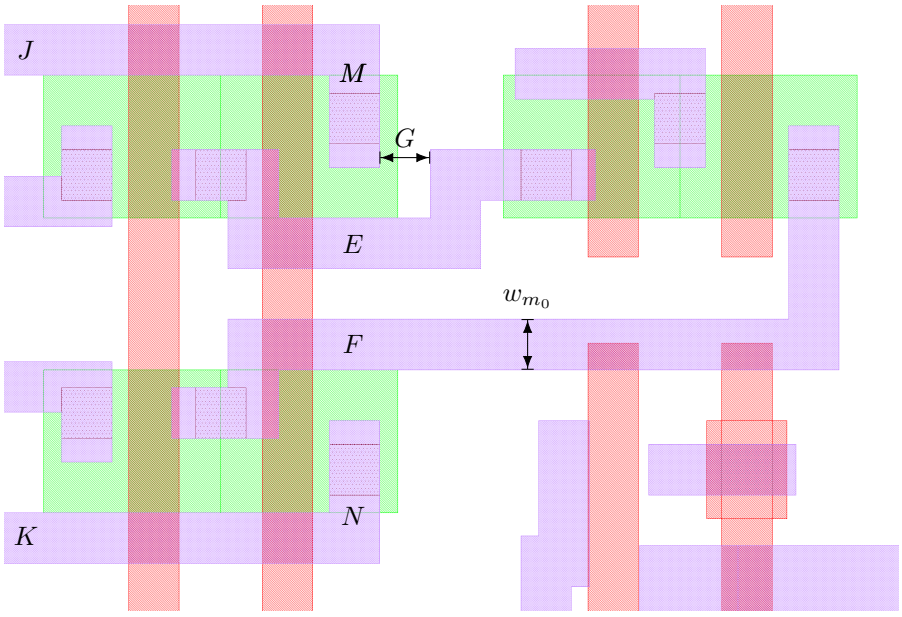


Figure 4.18: Metal m_0 wiring atop the base `Sky130Mux` generator layout

multiplexer, and their dimensions — in particular the polysilicon pitch — must be able to accommodate connections to upper layers. The outputs of each six-transistor group are

connected to the final mux on metal m_1, which is one layer above the local interconnect. The right side transistors are designated to be PMOS which, due to the generally lower charge mobility of P-type carriers, are typically wider. The PMOS transistors must also be placed in an N-well. Since they are grouped together, a single N-well can accommodate all of them and any additional margin is reduced.

The layout places an identical set of multiplexer control signals at the top and bottom, and an identical set of input signals at the left and right. The control signal inputs are staggered to aid access to wiring from, say, a router. Metal m_1 is used to connect opposite ends of the polysilicon columns to control signals in order to augment the poorer electrical conductivity of the long polysilicon structures. Then, in what is probably the single greatest flaw of this design, the inputs at the far left and far right of the layout must be connected together. This forces us to use a third metal layer, metal m_2. With its onerously sized via v_2 encapsulation rules, the m_2 wiring forces significant additional padding within the underlying structures.

The central difficulty in writing the generating program for this cell is trying to pre-empt constraint violations on the upper layers while laying out the lower layers. This is partly due to a lack of experience, but it is largely due to a bad layout model. Later approaches remedied this through better abstraction and still better approaches are envisaged for later work (Ch. 5).

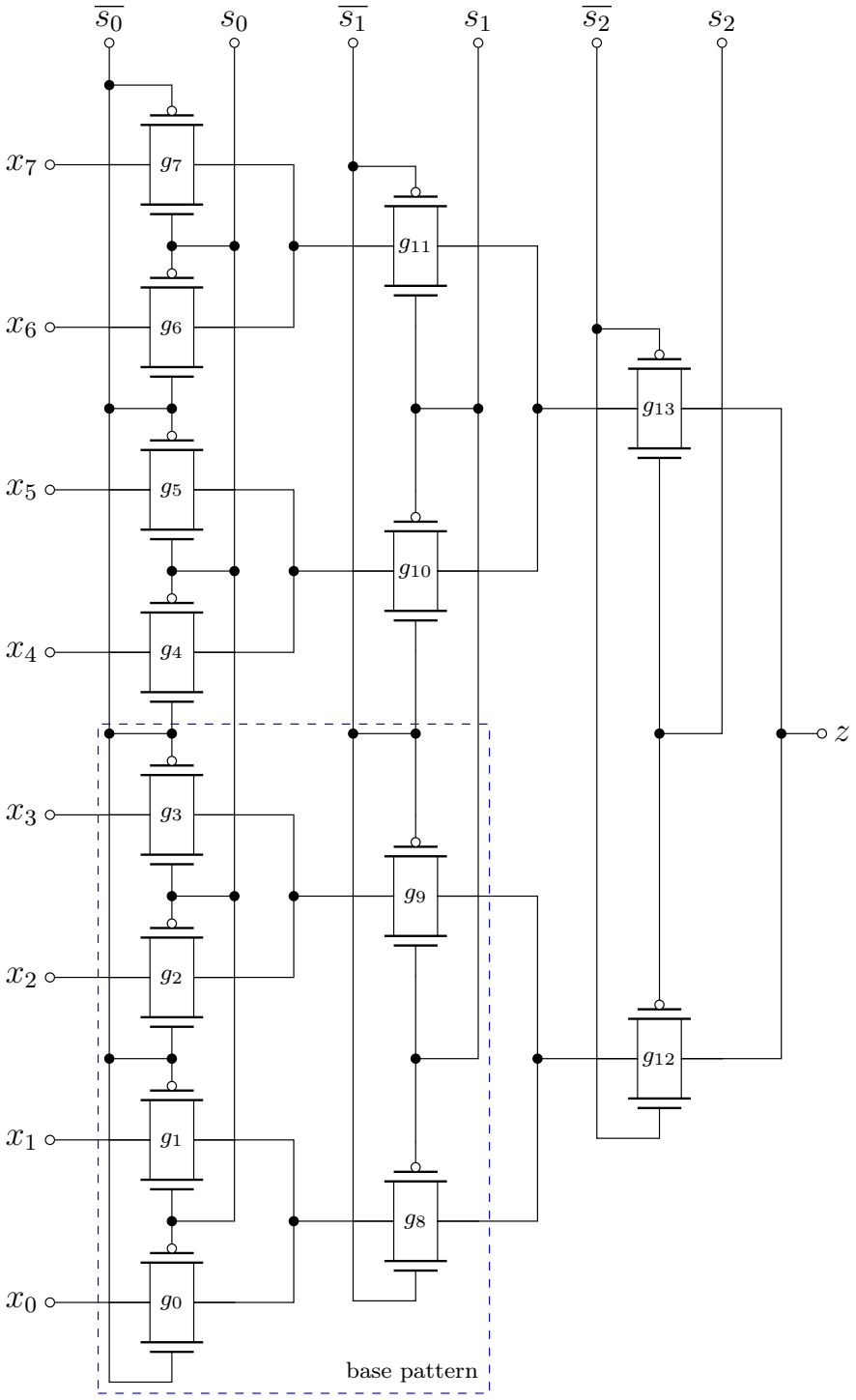


Figure 4.19: Idealised Sky130Mux circuit

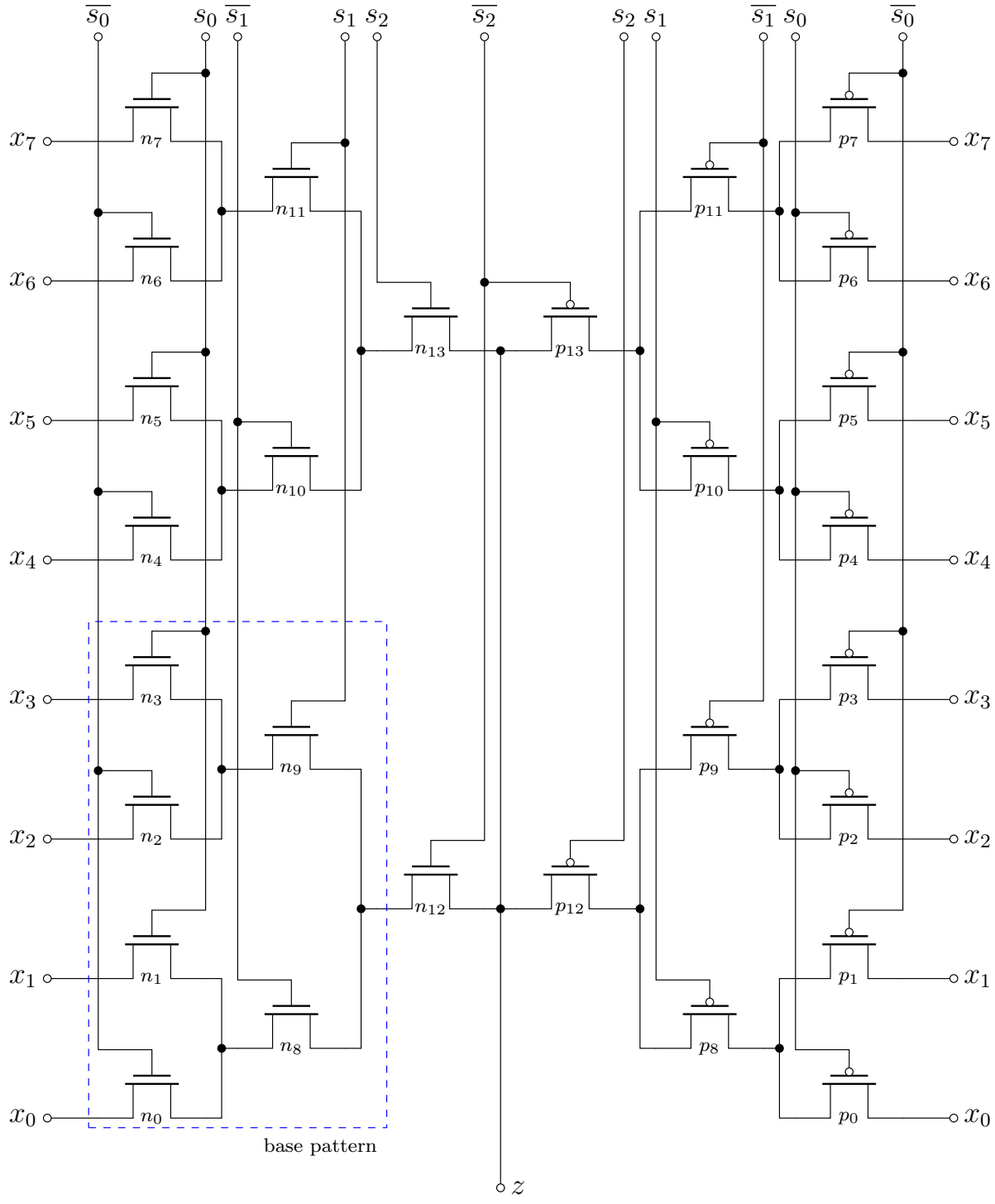


Figure 4.20: Actual Sky130Mux circuit

4.3.9 Hierarchical 4 : 1 Transmission-Gate Multiplexer for Global Foundries 180 nm

✓ DRC clean ⚠ LVS untested

atoms/gf180mcu_mux.h
atoms/gf180mcu_mux.cc

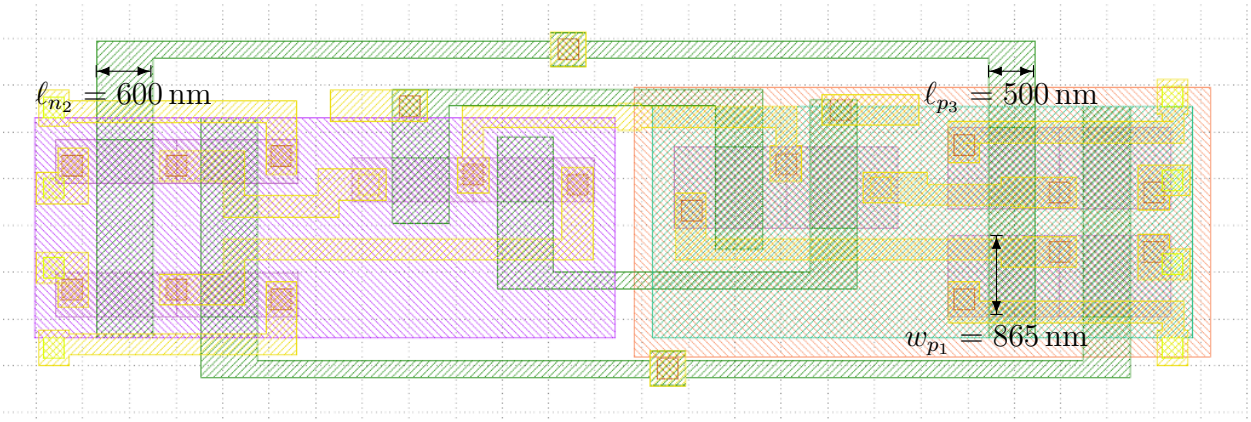


Figure 4.21: Hierarchical 4:1 transmission-gate multiplexer layout for Global Foundries 180 nm

Generated by `Gf180McuMux`, re-using methods from `Sky130Mux` with an a simple set of layer aliases and the Global Foundries 180 nm design-rule values.

The `Gf180McuMux` generator creates a 7-track standard cell for the Global Foundries 180 nm process, and is not used in any of the FPGA cells in later sections. Its layout is shown in Figure 4.21. The circuit produced by `Gf180McuMux` is a simple 4:1 subtree of the 8:1 transmission-gate mux shown in Figure 4.20.

The `Gf180McuMux` generator was written to demonstrate one of BFG's modalities for process portability, as introduced in Section 2.4. Recall that Global Foundries 180 nm (GF180) and SkyWater 130 nm (SKY130) are two processes with largely compatible layout topologies. The `Gf180McuMux` generator reuses the base pattern generator from `Sky130Mux` (Sec. 4.3.8) to produce a GF180-compatible layout. For this to work, the active `PhysicalPropertiesDatabase` needs to be loaded with GF180 rules instead of Sky130 (Listing 4.4), and a simple set of aliases needs to be defined to map the SKY130 layer names used by the `Sky130Mux` designer to their equivalent GF180 layer (Listing 4.3).

Listing 4.3: Creation of the SkyWater 130 nm to Global Foundries 180 nm layer aliases, from `dev_pdk_setup.cc`

```
373   db->AddLayerAlias("diff.drawing", "comp.comp");
374   db->AddLayerAlias("nsdm.drawing", "nplus.nplus");
375   db->AddLayerAlias("psdm.drawing", "pplus.pplus");
```

```
376   db->AddLayerAlias("poly.drawing", "poly2.poly2");
377   db->AddLayerAlias("nwell.drawing", "nwell.nwell");
378   db->AddLayerAlias("li.drawing", "metal1.metal1");
379   db->AddLayerAlias("met1.drawing", "metal2.metal2");
380   db->AddLayerAlias("met2.drawing", "metal3.metal3");
381   db->AddLayerAlias("met3.drawing", "metal4.metal4");
382   db->AddLayerAlias("licon.drawing", "contact.contact");
383   db->AddLayerAlias("mcon.drawing", "via1.via1");
384   db->AddLayerAlias("via1.drawing", "via2.via2");
385   db->AddLayerAlias("via2.drawing", "via3.via3");
386   db->AddLayerAlias("areaid.standardrc", "pr_boundary.pr_boundary");
387   db->AddLayerAlias("li.pin", "text.text");
```

Listing 4.4: Setting up a Global Foundries 180 nm cell instead of the SkyWater 130 nm one, from `main.cc`

```
93  class PhysicalPropertiesDatabase {
241 void Gf180McuMuxExperiment() {
242   // GF180MCU 7T MUX experiment
243   //
244   bfg::DesignDatabase design_db;
245   bfg::PhysicalPropertiesDatabase &physical_db = design_db.physical_db();

247   bfg::SetUpGf180Mcu(&physical_db);

249   std::string top_name = "gf180mcu_mux";
250   bfg::atoms::Sky130Mux::Parameters params;
251   bfg::atoms::Gf180McuMux generator(params, &design_db);
252   bfg::Cell *top = generator.GenerateIntoDatabase(top_name);

254   design_db.WriteTop(*top, "gf180_mux.pb", "gf180_mux.package.pb", true);
255 }
```

4.4 4-LUT CLB

✓ DRC mostly clean ✓ LVS pass

`tiles/lut_b.h`
`tiles/lut_b.cc`

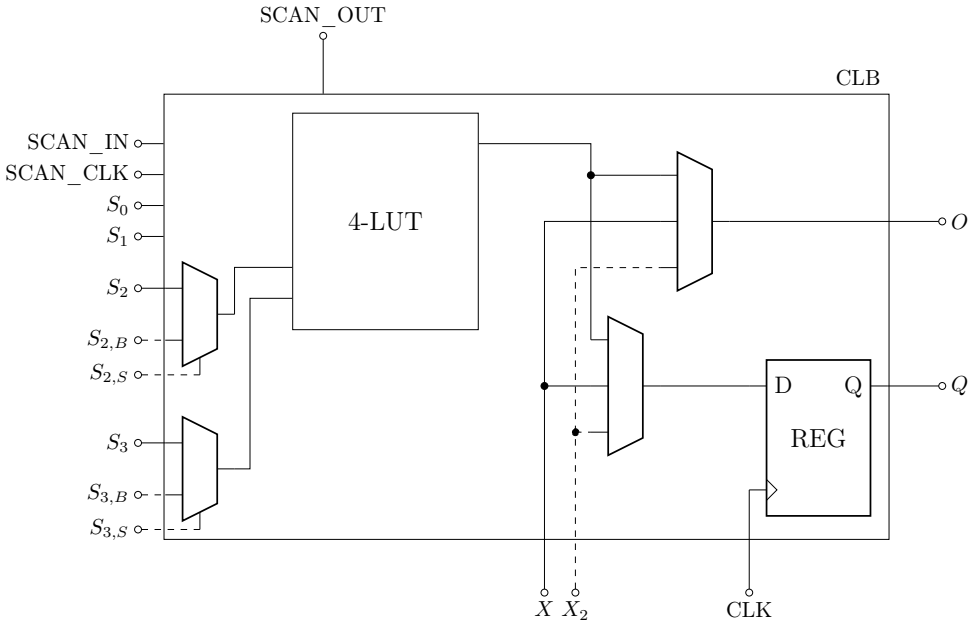


Figure 4.22: Architectural overview and optional features produced by the CLB generator

The 4-LUT-based CLB is the first meaningful *FPGA* component produced by BFG, implemented by the class `LutB`. This generator represents an early attempt at abstracting process-specific details into a *tile* (Sec. 2.4) by specifying high-level arrangements of basic elements and relying on the automated router (Sec. 2.7) to connect them. It is the generator's role to manage the complexity of adding optionality to the CLB through various optional architectural features. Its high-level architecture is shown in Figure 4.22. The cell is based arounda 4-LUT and implements features common to both academic and commercial fabrics alike (notably including the UltraScale from Chapter 3). It has:

- four input address lines for the 4-LUT, $S_i \mid i \in \{0, 1, 2, 3\}$;
- configuration input, clock and output signals;
- a bypass input X;
- a register, whose output is Q and whose input can be programmed by the FPGA bitstream to come from the LUT or the bypass input; and

- a combinational output O, whose input can be programmed by the FPGA bitstream to come from the LUT or from the bypass input.

In addition, the generator supports several architectural features which can be enabled or disabled by a corresponding parameter:

- the addition of a 2:1 multiplexer before the S_2 input, allowing an external $S_{2,S}$ to optionally select S_2 from $S_{2,B}$;
- the addition of a 2:1 multiplexer before the S_3 input, which works similarly; and
- the addition of a second bypass input X_2 and two 2:1 multiplexers so that each of the O and Q outputs can be programmed by bitstream $X2$ on top of X and the LUT output.

The timing performance of a CLB is determined by its worst-case read delay, since this determines how quickly a value can be retrieved and then registered within a cycle. The generator accordingly tries to minimise the worst-case delay by placing memories radially (or close to radially) around a central multiplexer complex. We opted for a scan-chain configuration architecture composed of flip-flops to 1) reduce wiring congestion, given the limitations of the SkyWater 130 nm process; and 2) simplify the design in the hope of de-risking the first attempt. It is common in modern fabrics to instead use SRAMs and frame-based configuration (which align with the wordlines of the SRAM array), but we leave that to future work.

The layout is composed as follows. There are three distinct parts: a left bank, a right bank, and a central grid (Fig. 4.23). The left and right banks are instances of the `MemoryBank` layout guide (Sec. 2.3.4), which manage an adaptive collection of rows (`RowGuides`). The rows manage alignment, rotation and automatic tap cell insertion as cell instances are added to them, leaving the generator to focus on the overall arrangement. Alternating rows are configured by each bank to be rotated 180°, replicating the standard-cell pattern, so that the supply rails and P/NMOS regions abut. The generator starts by assigning half of the memories to the left bank and half to the right, expecting to leave room in the bottom right for additional muxes and the application register. Then, input buffers are added for first three inputs $S_2 - S_0$ to the top row on the left bank: those corresponding to the least signifcant two digits of the LUT address are implemented with the split buffer cell, rather than regular buffers. As we shall see in Section 4.4.2, this is because they have the largest capacitive loads to drive. The memories are instances of `Sky130Dfxtp`, the buffers `Sky130Buf`, split buffers `130SplitBuffer`, and taps `Sky130Tap`. The central grid is an instance of the `CheckerboardGuide` helper. An instance of the `Sky130Mux` is placed in the bottom left and the top right.

As shown in Figure 4.23, the design is sufficiently abstract to depend only on the logical functionality of the constituent shells, and not any particular generator (that is, as long as the cells correctly stack in rows and expose reachable ports). We can thus achieve process

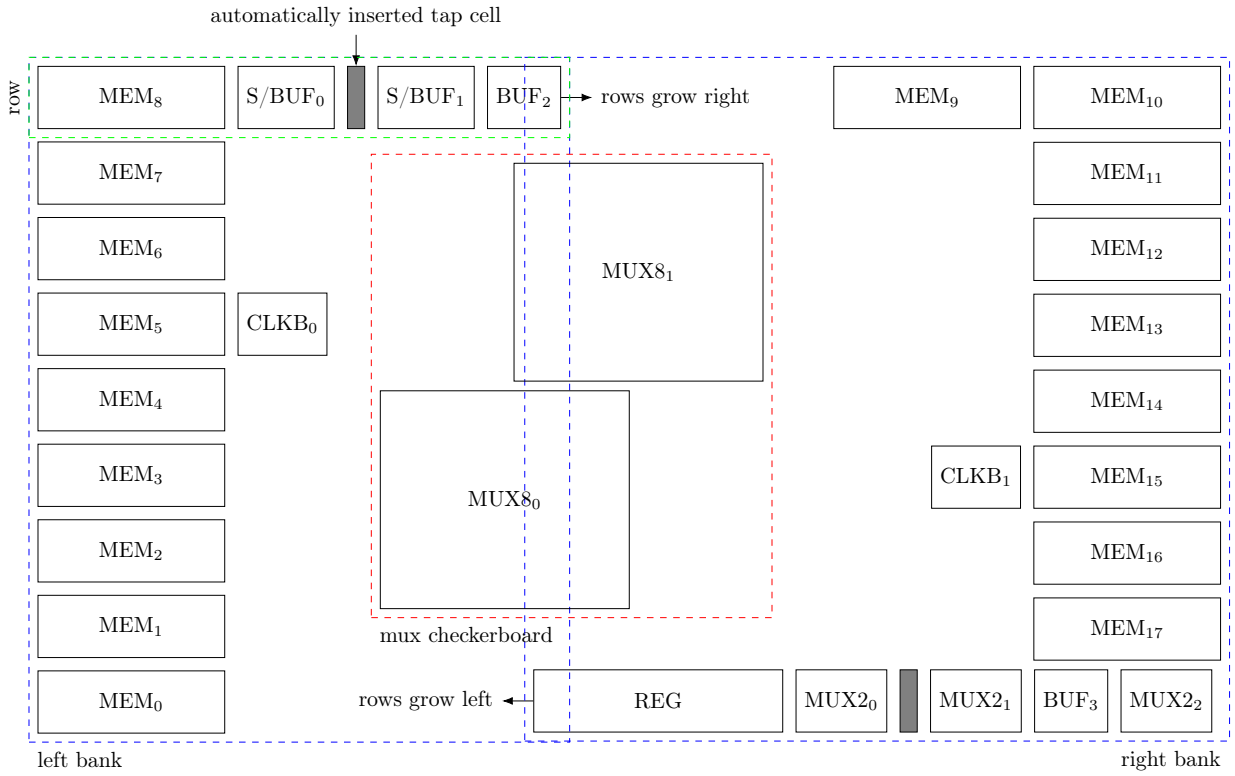


Figure 4.23: Layout strategy for the 4-LUT CLB

portability through the second modality described in Section 2.4. Each cell can be swapped for a logically-equivalent cell generator for a different process and still generally work, exactly as in the case of standard cell designs. The idea is that any particularities between different process can be addressed through tuning. In reality there are more particularities than expected, and the `LutB` code often has to query PDK rules for spacing information. But this, too, can eventually be abstracted. For instance, the class can test that multiplexers are not too tall, and create additional rows otherwise.

Once the three layout components are assembled, a margin is added, and the layouts are stitched together with the rows lined up. Any excess space in the rows is automatically filled with decap cells. A routing grid is generated over the top, all existing shapes are added as obstacles (Sec. 2.7), and all of the necessary wires are laid. The margin is a parameter that the designer controls; it allows more or less room to given to the router in order to assess congestion. In its default configuration the generator produces the layout of Figure 4.24, which corresponds to the detailed of Figure 4.25. The default configuration enables no additional inputs. Listing 4.5 shows the corresponding configuration file.

Listing 4.5: Default configuration file for the `LutB` CLB generator

```
add_s2_input_mux: false
```

```
add_s3_input_mux: false
add_third_input_to_output_muxes: false
disable_routing: false
```

If we instead opt to add the second bypass input $X2$ as a "third input to the output muxes", the generator would output the layout of Figure 4.26. We shall see examples of the other options in Section 4.6.

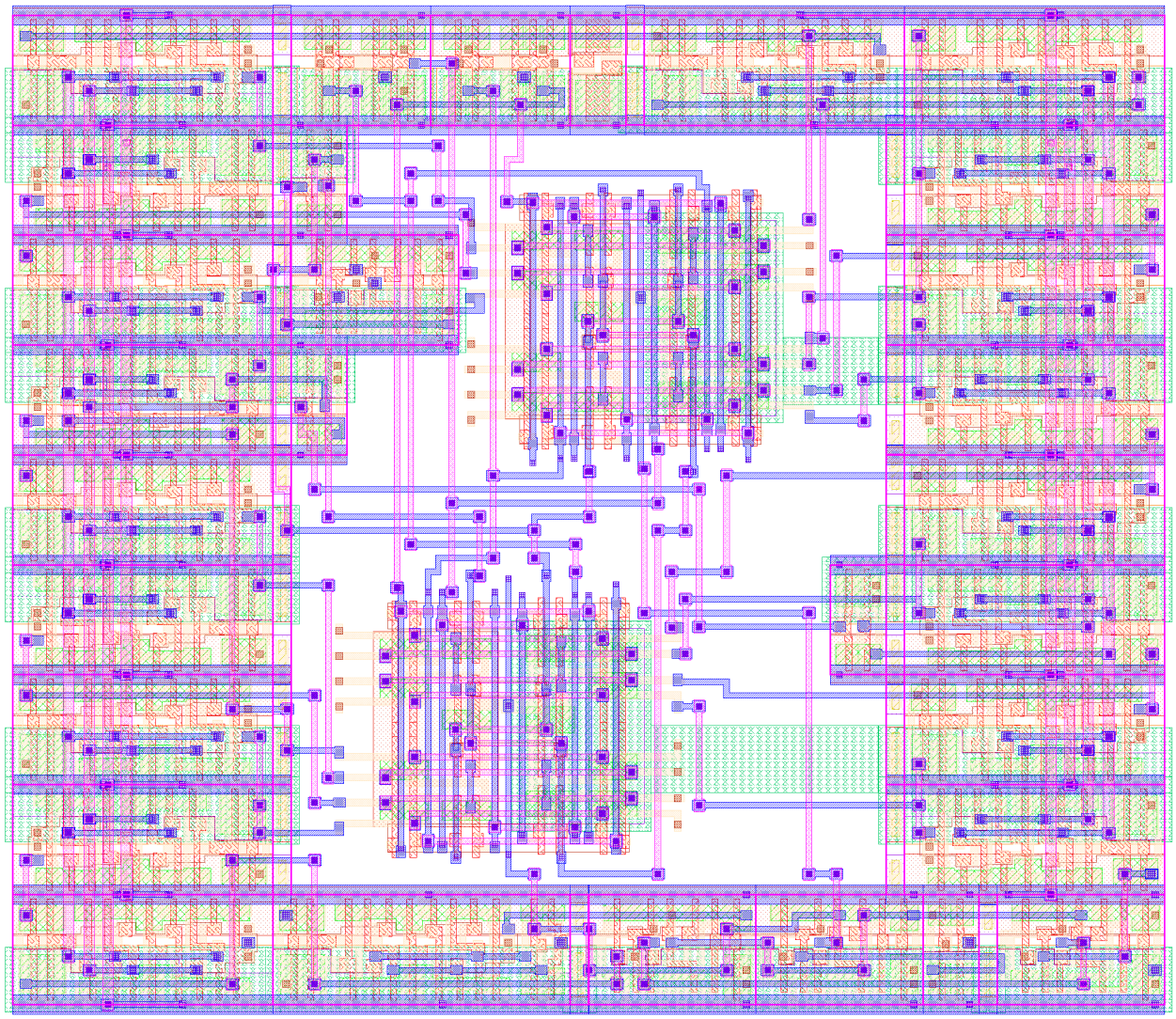


Figure 4.24: CLB layout generated by `LutB` in its default configuration

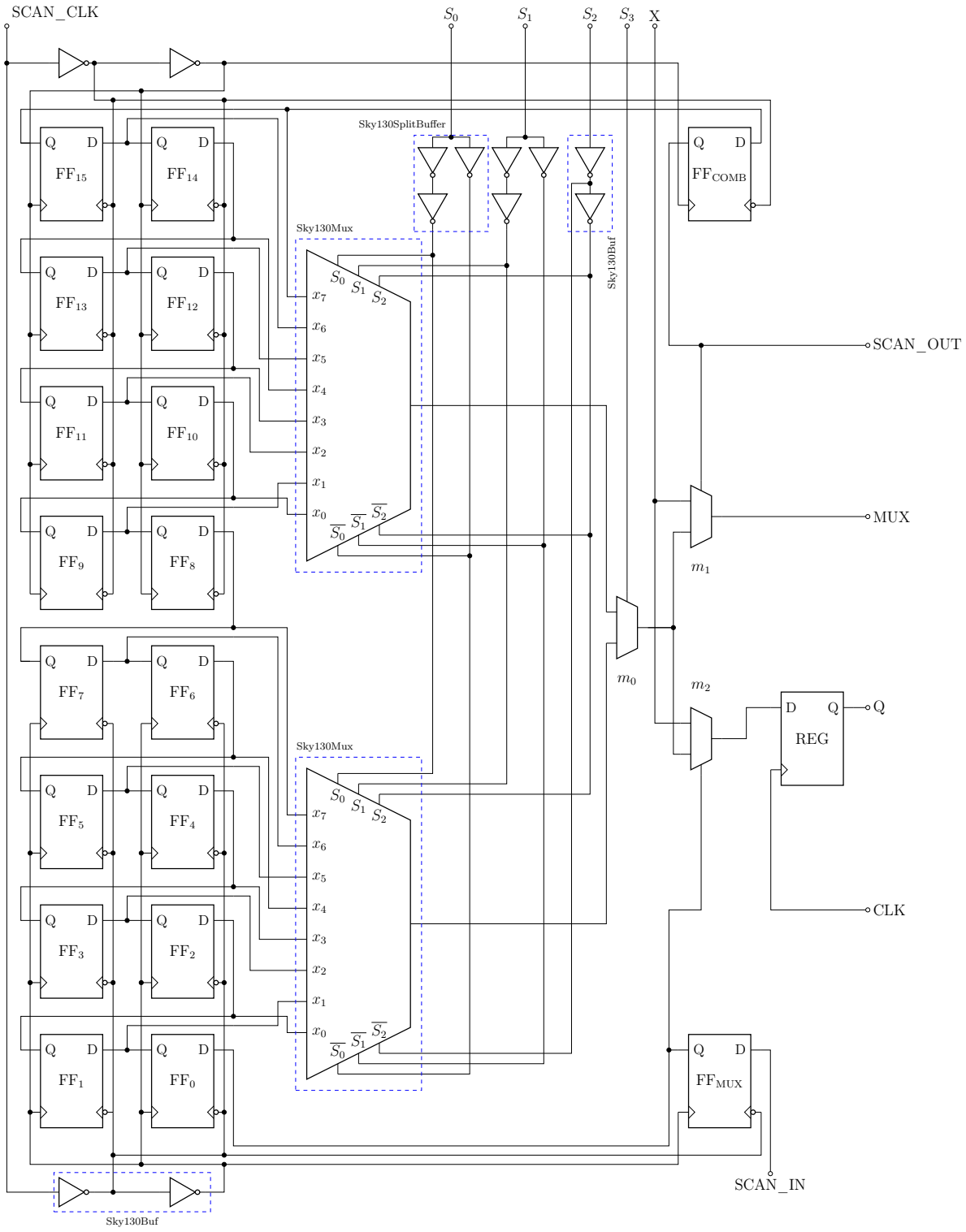


Figure 4.25: CLB circuit with default parameters
No additional bypass inputs; S_2 and S_3 are not split.

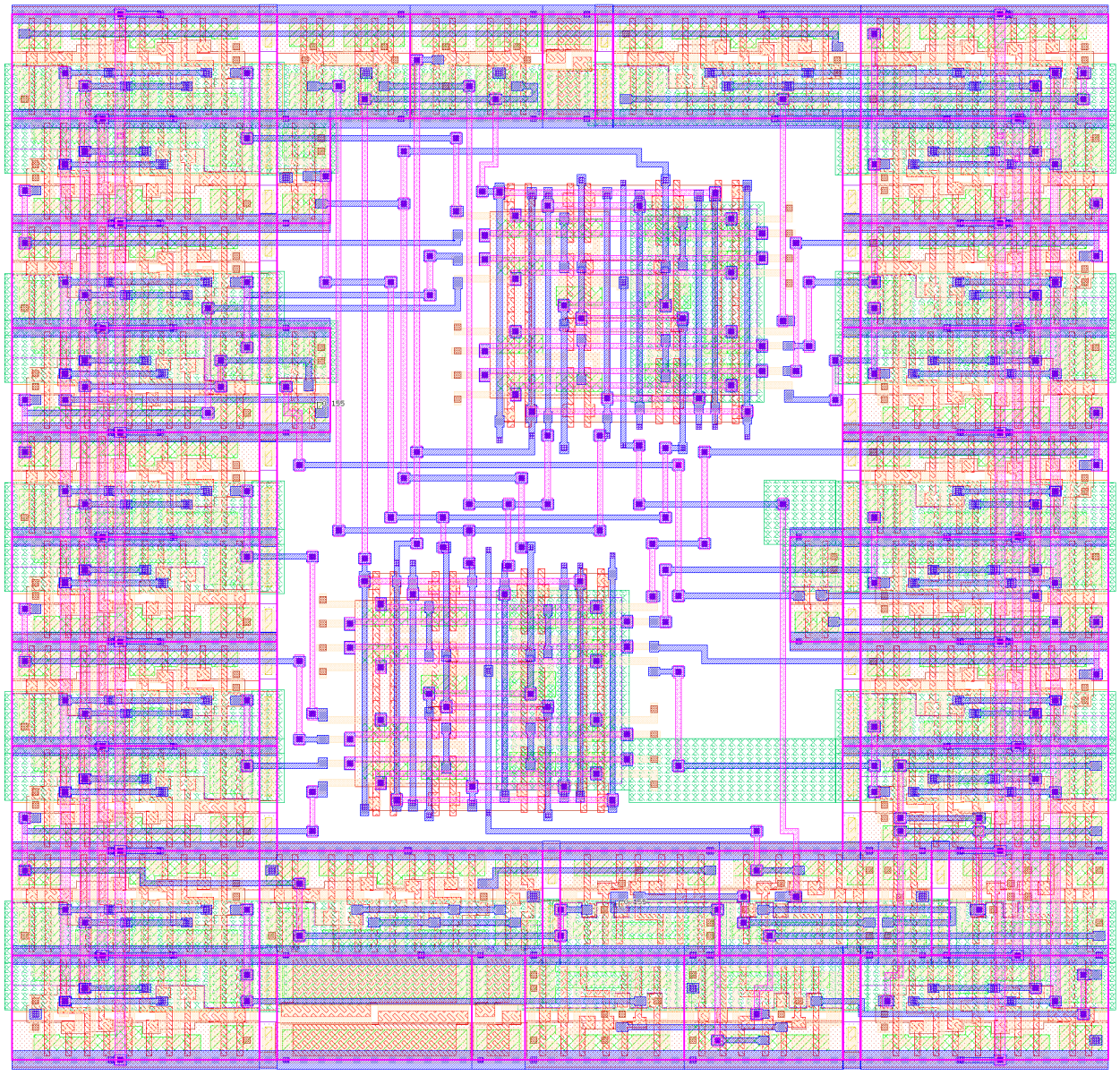


Figure 4.26: 4-LUT CLB generated with the option for a second bypass input

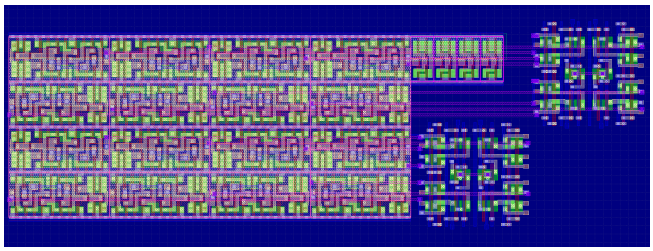
4.4.1 Earlier Attempts

Before `LutB` there was `LutA`, our first attempt at writing the CLB generator. This generator was parametrised to support k-LUTs of size $k \in \{4, 5, 6\}$. Design centred on the central k parameter and how that could easily be implemented. A natural first approach is to make a bank of 2^k memories and try to read from it, like in Figure 4.27a. As we have seen, the read delay of a LUT is dictated by the worst-case, and so to try and balance the read delays between two sides we arrived at Figure 4.27b.

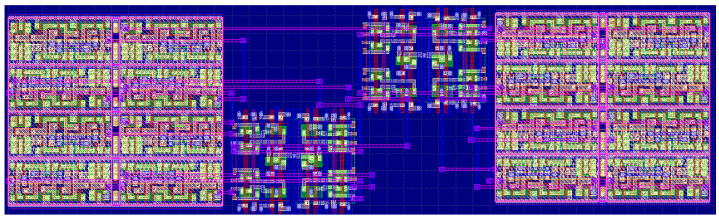
Once `LutA` could trivially scale with k we could produce Figures 4.28a – 4.28c. Two problems quickly emerged:

1. the longest wire path from any memory to the output, and hence the worst-case read delay, seemed to be penalised at low k; and

2. stacking memories in columns created a lot of wiring congestion when trying to route out from each memory to the multiplexer inputs.



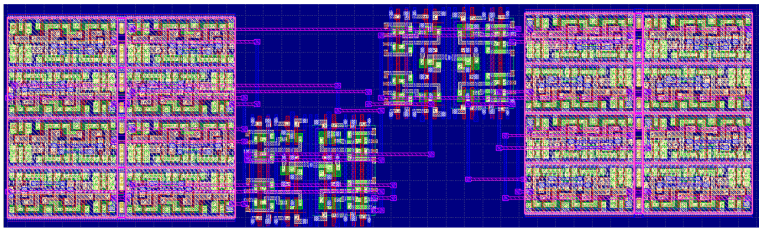
(a)



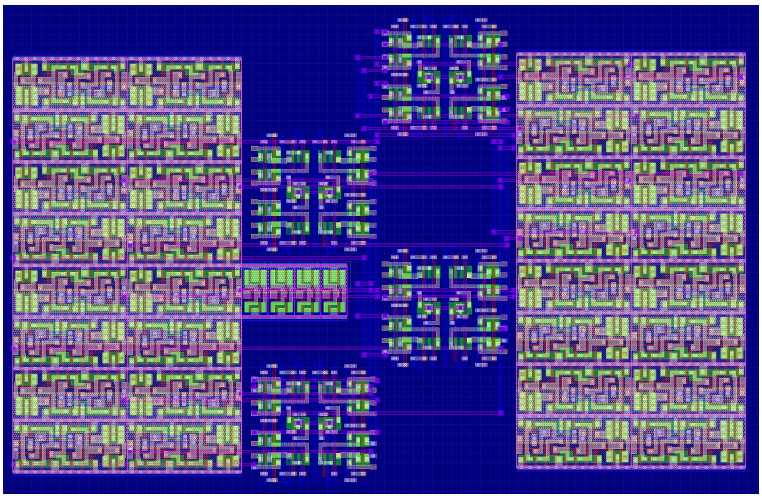
(b)

Figure 4.27: Early 4-LUT layouts generated by `LutA`

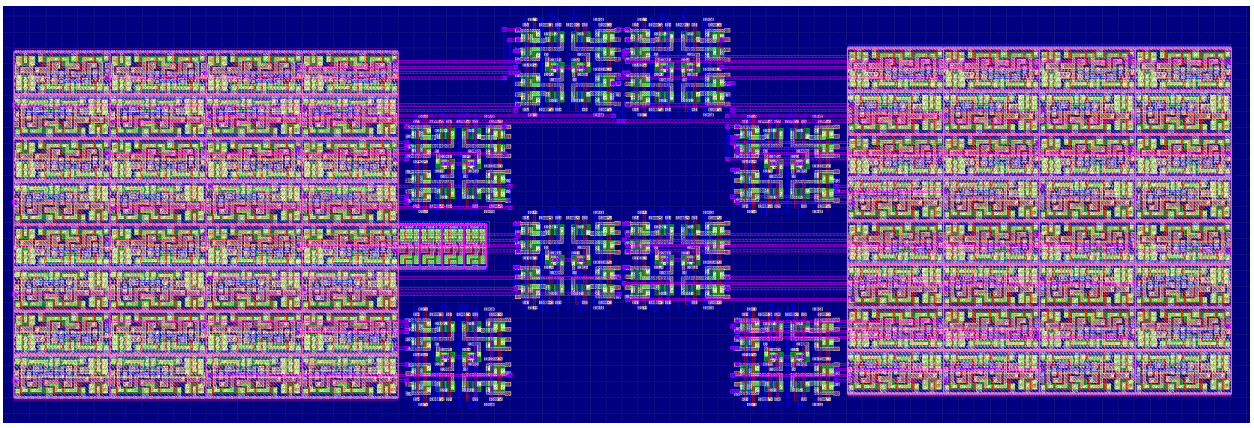
As we saw in the previous section, `LutB` was devised to centralise the multiplexers and circumscribe them with memories as a direct response to this foolishness. We can still generate LUTs for $k > 4$ by double the number of existing `LutB` instances: two for $k = 5$, four for $k = 6$, and so on. This has the benefit of keeping the read delay within each group of 16 memories closer to optimal and gives us some measure of control over the overall aspect ratio of the layout.



(a)



(b)



(c)

Figure 4.28: An early scaling pattern for k-LUT CLBs generated by `LutA` for $k = \{4, 5, 6\}$

4.4.2 Tuning

A significant advantage of a silicon compiler is that we can tune designs, using high-fidelity simulations of finished layouts, in a way that is both more automated than in the custom design cycle and simply not possible with fixed standard-cells. Standard-cells, which must be chosen from among a few discrete types, provide no mechanism for circuit- or layout-level adjustment. Such is the double-edged sword of their ability to express vastly complex digital designs. Custom design absolutely requires users to simulate and tune their circuits, but it is a manual process. Silicon compilers expose many precision parameters per circuit and can be built to integrate circuit tuning into the generation of the circuit itself. What's more, since the generators are regular pieces of software, they can incorporate any number of existing methods to do so: reinforcement learning; dynamic programming; integer, mixed-integer and quadratic programming; and so on. Although BFG does not yet implement automatic self-tuning, we will demonstrate how manually doing so can still inform generator improvements that have a significant impact on the performance of output designs.

Before we can proceed with tuning, we must ensure that a given design is DRC-clean (there are no[4] violations of foundry layout rules) and passes an external LVS check (meaning they implement the circuit that BFG claims they do). Then we perform parasitic extraction using a third-party tool. These tools predict the parasitic resistances, capacitances and inductances that would otherwise be present in a manufactured chip based on its final layout, which are then used to augment the initial circuit description. The resulting circuit model can much more accurately capture the performance of the IC device post-manufacture.

Accordingly, once the default CLB generated by `LutB` was DRC-clean and could pass LVS, we measured the worst-case read-delay and average power use of the circuit over a range of transistor sizes in the read path. Note that there is already a balance to be struck in the experimentation: if we regenerate the layout and re-run parasitic extraction at each candidate transistor size, we will have more accurate parasitic values but spend significantly more CPU time and limit the number of points we can measure; if we start with a fully extracted circuit and simply modify the circuit model with each test point, we will be able to run many more tests but with degraded fidelity. We opt for the latter to guide the tuning process and the former to come up with final performance measurements when needed (Sec 4.4.3).

The CLB read path consists of the CLB input buffers, the transmission gates in the central multiplexers, and the driving buffer in the memory storage cell. These experiments yielded two key insights. First, the output layout works as intended in simulation. Second, we should use split buffers (Sec. 4.3.2) for the two least-significant LUT address inputs. These are the ones that turn on the largest layers of the multiplexer tree and so have high capacitive loads. With a regular buffer generating both the control signal and its complement, one is significantly delayed from the other, and the transmission gates spend a lot of time turning

[4]There are classes of rule violations that are routinely ignored by even the foundry themselves, so really what we mean is that the design is "close enough" to having no rule violations.

on and off. With the split buffer, these paths are separated and their respective delay is much less.

The simulation was set up as follows. An alternating sequence of low and high values was loaded into the configuration chain, simulating the scan-chain load itself. Then two tests were conducted: a) a sequential read of every LUT index from 0 to 15; and b) a sequence of 100 random reads set up to always ensure a value transition at the output. Then the configuration bitstream was inverted and the two tests were repeated. The presented worst-case read delay and average power consumption were taken across all four sets of results.

With the split buffer swapped in, we returned to sizing transistors in the read path. Figures 4.30 and 4.29 show the effect of sweeping transistor sizing in: the split buffers (Figs. 4.30c, 4.30d); the memory cell output buffer (Figs. 4.30a, 4.30b); and the transmission gates of the `Sky130Mux` multiplexers (Figs. 4.29a, 4.29b).

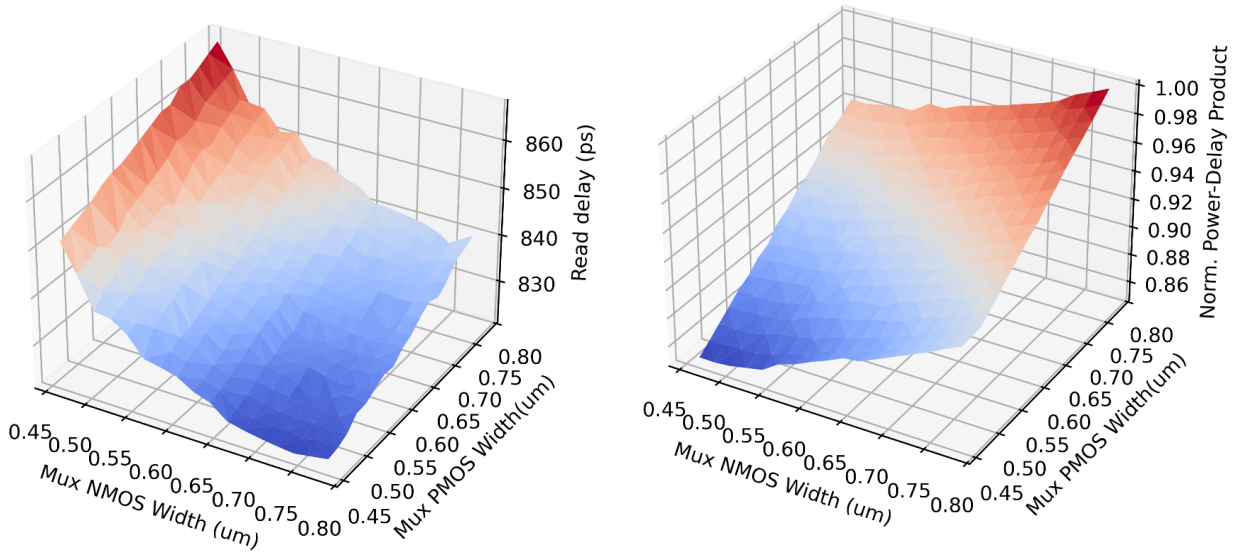
The notable results are:

1. Increasing the flip-flop buffer's NMOS transistor width decreased worst-case read delay, but increasing the PMOS width *decreased* power. This suggests that while it was not the bottleneck to read delay, driving the output line high faster reduced the total time transistors spent in linear mode versus in saturation or off.

2. In the hierarchical transmission gate multiplexers of `Sky130Mux`, *smaller* PMOS than NMOS transistors yielded *better* read-delay, but for the power-delay product overall, minimising both was best.

3. Increasing the S_0, S_1 address line buffer sizes reduced delay as expected, but the power-delay trade-off yielded a local-minimum somewhere in middle of the range. These values were selected for the final configuration.

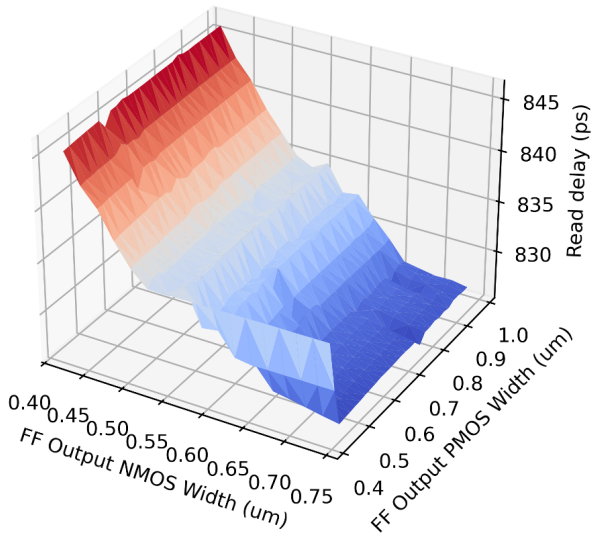
With tuning complete, we now know that we can achieve lower delay or lower power by tweaking transistor sizes, but not both.



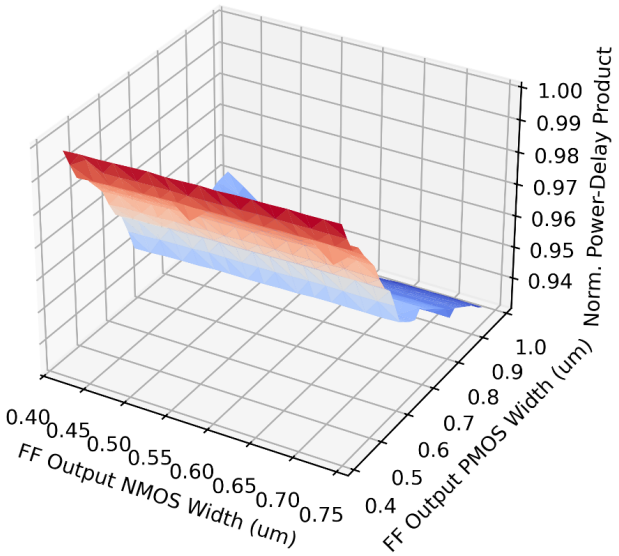
(a) Read delay vs `Sky130Mux` gate sizing

(b) Power-delay product vs `Sky130Mux` gate sizing

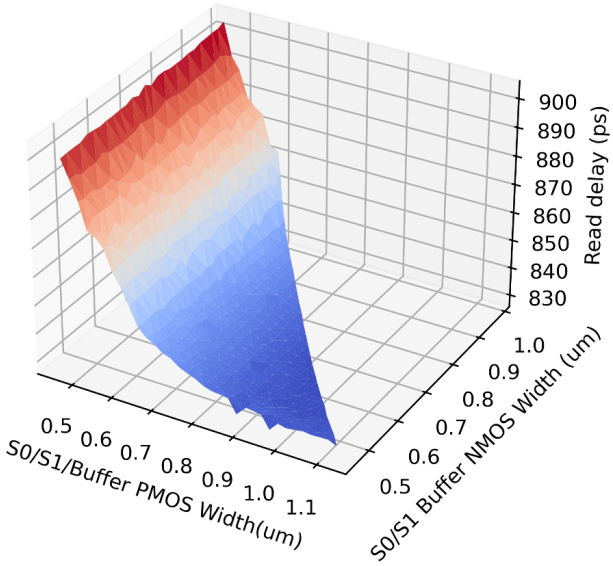
Figure 4.29: Tuning `Sky130Mux` gate sizing



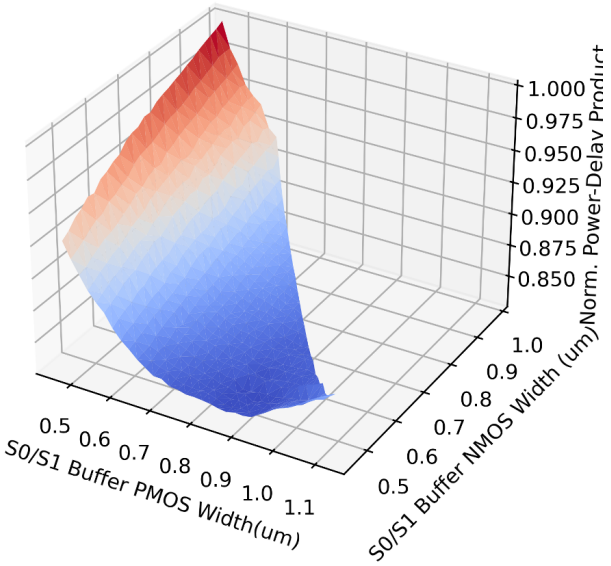
(a) Read delay vs driving flip-flop buffer sizing



(b) Power-delay product vs driving flip-flop buffer sizing



(c) Read delay vs S_0, S_1 input buffer sizing



(d) Power-delay product vs S_0, S_1 input buffer sizing

Figure 4.30: Tuning CLB read-path buffer sizing

4.4.3 Performance

Since a key hypothesis of this work is that writing a silicon compiler is an advantageous way to build FPGA fabrics, one that balances the performance of custom design with the flexibility of standard-cell based eFPGAs, we compared the performance of the `LutB` CLB to several logically-identical standard-cell implementations.

Experimental Setup

We implemented the CLB architecture of Figure 4.22, with default options (none of the dashed lines), in Verilog. Listing 4.6 shows the module as implemented with the parameters defaults set to their simulated values. We then used one of two digital VLSI tool flows to implement it (synthesise and then place-and-route) using one of two standard cell libraries for SkyWater 130 nm. We performed parasitic extraction and then ran the same simulations described for tuning in Section 4.4.2 to measure worst-case read delay and average power consumption. The digital flow tools were allowed to infer a scan chain and therefore were free to re-arrange the order of the memories. The scan-enable signal, not present in the BFG design, is a hint to use scan-chain flip-flop cells. The simulated bitstream was modified for each to ensure that the output toggled as BFG's did. Table 4.7 summarises these configurations. Innovus, Quantus, and Spectre are tools from Cadence Design Systems, Inc. Calibre is by Siemens. LibreLane is an open-source system built around OpenROAD, discussed in Section 1.7.1.

Table 4.7: Standard-cell flows benchmarked against BFG

Reference	Skywater 130 nm standard-cell library	Tools		
		Synthesis, place-and-route	Parasitic extraction	Simulation
LibreLane HD	FOSS High-Density	LibreLane 3.0	Calibre v2022.2	Spectre 24.1
Innovus HD	FOSS High-Density	Innovus v22.11	Quantus 22.1	Spectre 24.1
Innovus SCL	Commercial	Innovus v22.11	Quantus 22.1	Spectre 24.1

Listing 4.6: Verilog implementation of the default `LutB` CLB configuration

```
1  module clb #(
2      parameter INPUTS=4, MEM_SIZE=1<<INPUTS, CONFIG_WIDTH=1
3  ) (
4      input clk,
5      input [INPUTS-1:0] addr,
6      input x,
7      output mux,
8      output q,
9      input scan_clk,
10     input scan_en,
11     input [CONFIG_WIDTH-1:0] scan_in,
12     output [CONFIG_WIDTH-1:0] scan_out
```

```
);
reg q_control = 0;
reg [MEM_SIZE-1:0] mem = 0;
reg mux_control = 0;
reg q_reg = 0;

wire z = mem[addr];
always @(posedge clk) begin
  q_reg <= q_control ? x : z;
end
assign mux = mux_control ? x : z;
assign q = q_reg;
assign scan_out = mux_control;

generate
    genvar i;
    for (i=1; i<(MEM_SIZE/CONFIG_WIDTH); i=i+1) begin
        always @(posedge scan_clk) begin
            if (scan_en) begin
                mem[(i+1)*CONFIG_WIDTH-1:i*CONFIG_WIDTH] <= mem[
                    (i)*CONFIG_WIDTH-1:(i-1)*CONFIG_WIDTH];
            end
        end
    end
endgenerate

always @(posedge scan_clk) begin
    if (scan_en) begin
        q_control <= scan_in;
        mem[CONFIG_WIDTH-1:0] <= q_control;
        mux_control <= mem[MEM_SIZE-1:MEM_SIZE-CONFIG_WIDTH];
    end
end

endmodule
```

Results

The performance results of BFG and each of the three comparison standard-cell implementations are shown in Table 4.8. The layouts generated for each are shown in Figure 4.31. First, we note the difference in just *synthesis* performance across the three standard-cell digital flows. They each synthesised a decoder to read the flip-flop chain (as opposed to an explicit multiplexer structure). Innovus re-arranged the scan-chain into what it deemed to be an optimal order, LibreLane did not. The Innovus HD, Innovus SCL and LibreLane HD flows implemented the circuit with 46, 37 and 89 cells, respectively. The latter included 37 buffers for timing repair alone. The delay measurements *without parasitics* for each flow provide some measure of the intrinsic or *idealised* delay of each circuit. This suggests that, before layout is even considered, the circuit used as the basis is a significant factor in the measured performance.

Parasitics add significant error to the delay measurements; fully 50% of the delay in the BFG circuit is attributable to parasitic capacitances and resistances in the layout. If anything, we can look to the penalty of including parasitics over the idealised circuit as a

Table 4.8: Performance of BFG CLB vs. synthesised CLBs

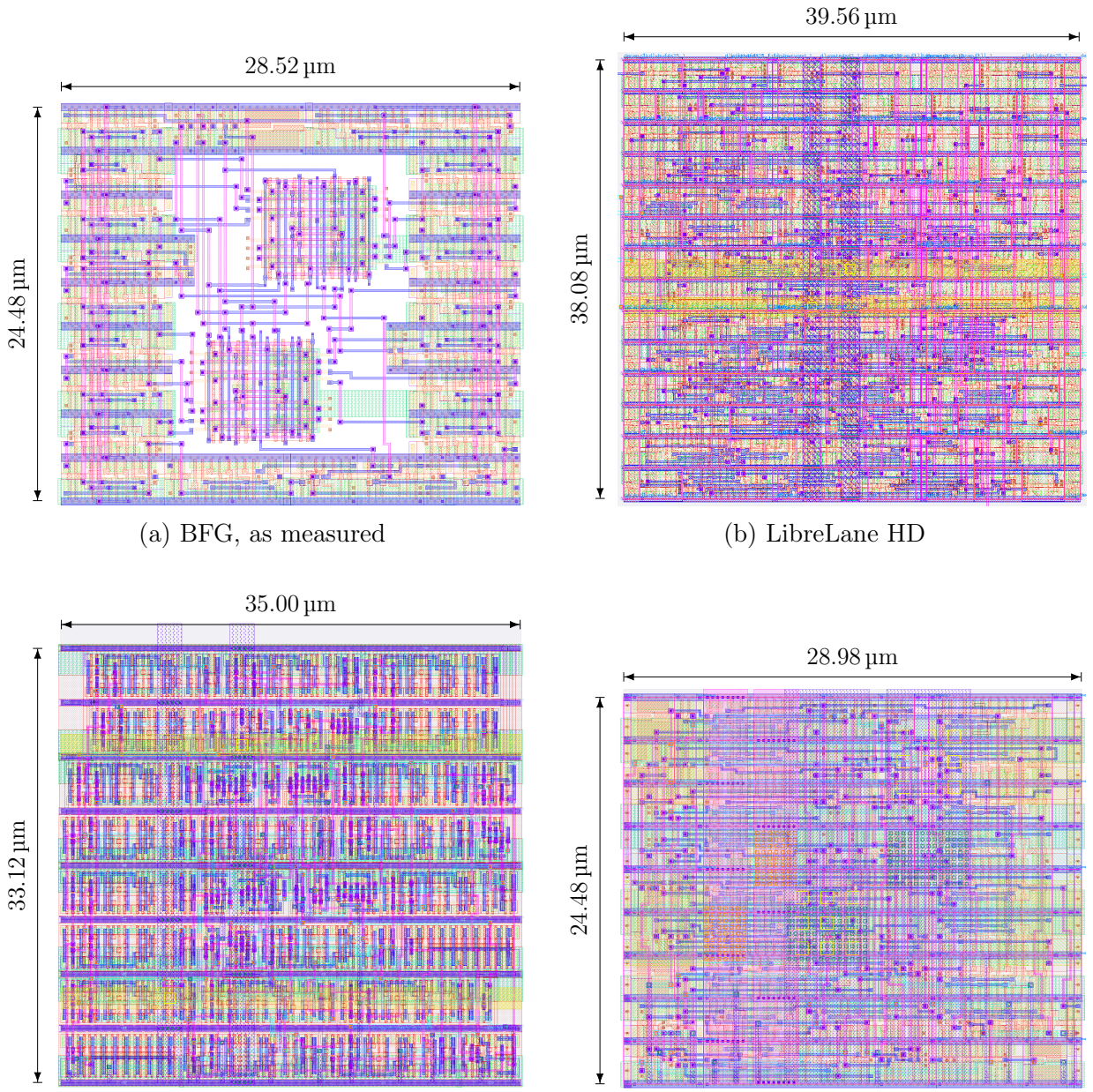
Metric	LibreLane HD	Innovus HD	Innovus SCL	BFG	Gain
PAR Density (%)	81.6	99.8	96.6		
Area (μm^2)	1506.4	709.4	1159.2	698.9	-39.7%
Max. delay (ps) (no parasitics)	1120.8	610.0	540.1	437.5	-19.0%
Max. delay (ps) (w/ parasitics)	1925.2	-	955.6	867.1	-9.3%
Power (µW) (w/ parasitics)	319.7	-	242.5	183.6	-24.3%
Power, Delay, Area Product	3.45	-	1.00	0.414	-58.6%

measure of the quality of the layout itself. For BFG it adds a 98.2% penalty, for Innovus SCL 76.9%, and for LibreLane HD 71.8%. This merely highlights the fact that standard cells themselves are very well optimised analog/mixed-signal circuits: BFG's layout has much room for improvement, as any experienecd layout designer will readily note. We believe that the biggest opportunity for optimisation is around the transmission-gate multiplexers.

Next, note the achieved Place-And-Route (PAR) density of each of the standard-cell runs, which measures the sum of the areas of all the placed standard-cells as a fraction of the total design area. The Innovus-driven layouts achieved 96.6%–99.8% density, suggesting that their layouts approach the limit of what can be achieved with standard-cells. At 100% density there is no room to improve the layout area since the standard cells themselves are immutable, meaning that the only electrical advantages can come from shorter wiring between cells, perhaps through a different arrangement. The most we could achieve with the open-source LibreLane flow was a venerable 81.6% density. Beyond this, its router would complain of too much congestion and give up. This is unsurprising given that synthesised netlist in this flow has nearly twice as many cells as in the other two.

At 99.8% density, the output of the Innovus run with the open-source high-density library (Innovus HD) is approximately the same size as the BFG layout: BFG is only about 1.4% smaller. The high density cells are *designed* to save area at the cost of performance, so it is expected that at near total utilisation (100% density) the design would be very compact. However, we could not get the Innovus HD flow to produce DRC-clean layout, and as a result we could run neither LVS nor parasitic extraction. Just comparing the idealised delays of the circuits, BFG is 28.3% faster.

The Innovus SCL flow produced a larger but faster circuit than the Innovus HD flow, and both a smaller and faster circuit than the LibreLane HD flow. Compared to the Innovus SCL flow, BFG's design is 39.7% smaller, 9.3% lower delay and 24.3% lower-power. We



(a) BFG, as measured

(b) LibreLane HD

(c) Innovus SCL

(d) Innovus HD

Figure 4.31: Layouts produced by each of the four CLB generation flows
(Not to scale.)

define the *power-delay-area product* — the product of power, delay and area all measured with parasitics — as a figure of merit to capture the three dimensions of performance. This is shown in Table 4.8 with results normalised to the Innovus SCL flow, since it is the best

standard-cell implementation we have. Overall, BFG achieves a 58.6% better power-delay-area product than the next best implementation.

4.5 Carry Chain Bit

⚠ DRC untested ⚠ LVS untested

`atoms/sky130_carry1.h`
`atoms/sky130_carry1.cc`

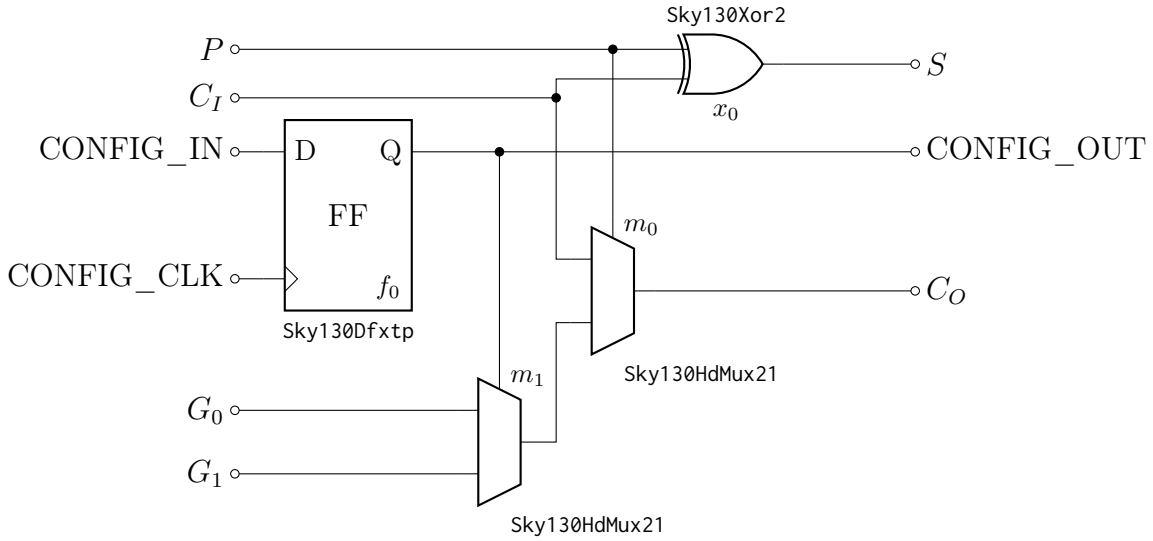


Figure 4.32: Carry bit circuit

The `atoms::Sky130Carry1` generator implements a common CLB carry-chain structure based on the work by [40]. Figure 4.32 shows the corresponding circuit: each cell implements a single bit of an N-bit summation. As we shall see in the next section, it is expected that this structure will be fed by two look-up tables: one to store the propagate P signal for a 1-bit sum, and another to store the generate bit G. The carry logic then computes the sum S and carry-out C_O according to P, G and the carry-in C_I as

$$S = P \oplus C_I$$
$$C_O = PC_I + G$$

where the latter is computed using a multiplexer instead of explicit logic. The cell adds a second generate input G_1, which can be selected by an additional configuration bit. G_1 is typically connected to a bypass input and affords flexibility in splitting the carry chain; G_0 is typically connected to the output of a LUT as desribed earlier.

The layout of the cell is shown in Figure 4.33. We use instances of `Sky130Dfxtp` for the scan-chain configuration bit (Sec. 4.3.3), `Sky130HdMux21` for the two 2:1 multiplexers (Sec. 4.3.6); and `Sky130Xor2` for the two-input exclusive-or gate (Sec. 4.3.7). The cells are arranged in a single `RowGuide`, which automatically inserts the tap cells. Wiring is generated to connect

the gates on a single layer, adhering to regular metal m_0 track width and pitch, without needing to invoke the router. The carry-in C_I and carry-out C_O are placed opposite each other at the western and eastern edges so that they can connect by abutment to neighbouring instances of the same cell.

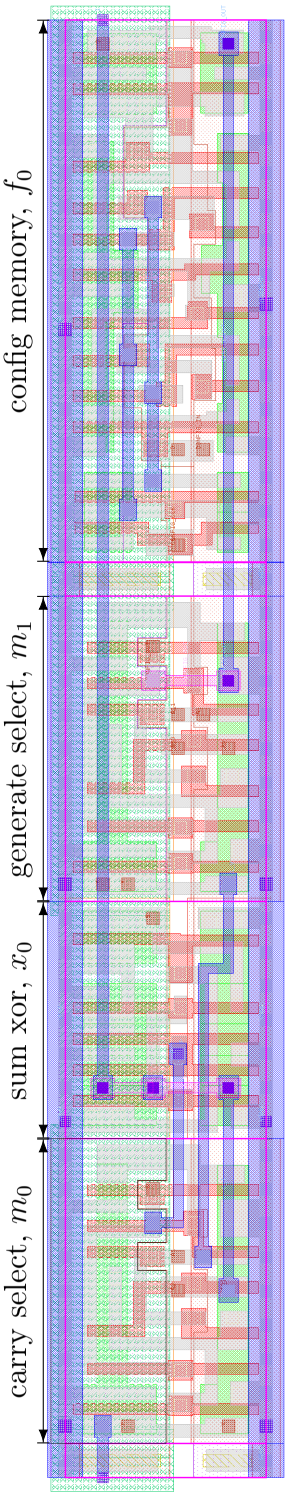


Figure 4.33: Layout for a single bit of the carry chain, generated by `atoms::Sky130Carry1`

4.6 S-44 CLB

× DRC errors ⚠ LVS untested

`tiles/s44.h`
`tiles/s44.cc`

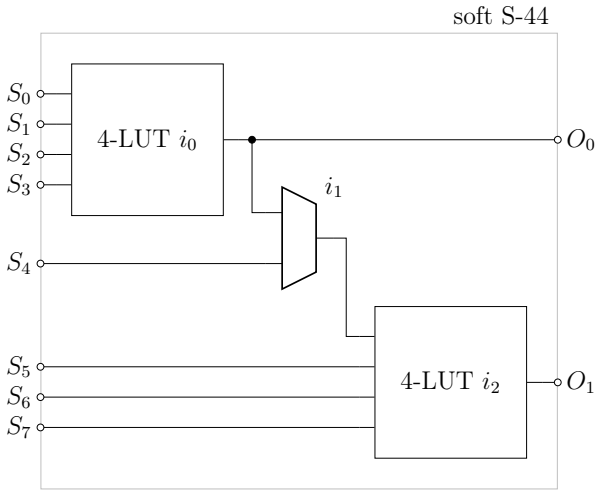
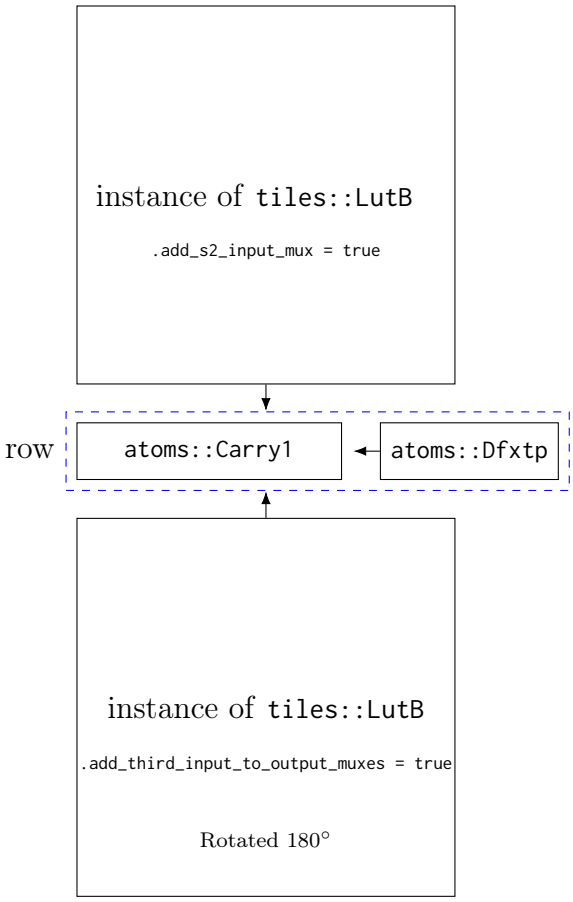


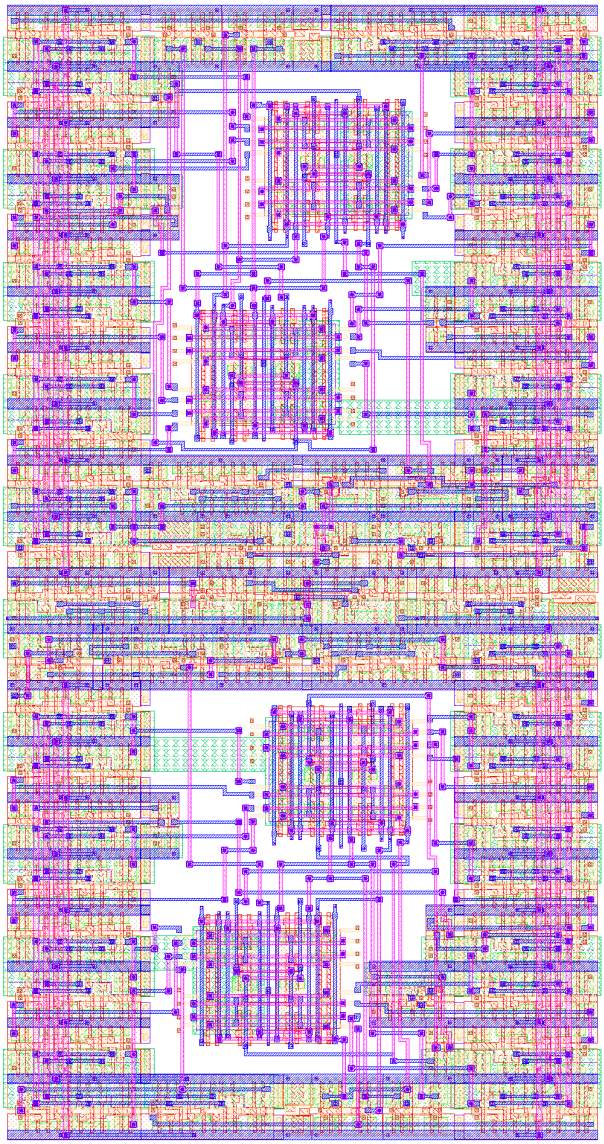
Figure 4.34: A soft S-44 LUT daisy-chains two 4-LUTs

In [28], the authors show that two 4-LUTs, configured in the "S-44" configuration shown in Figure 4.34, can achieve performance close to a 6-LUT while being significantly smaller and consuming less power (two 4-LUTs together have $2 \times 2^4 = 32$ entries, a single 6-LUT has $2^6 = 64$ entries). The so-called *soft* S-44 configuration introduces the multiplexer i_1 so that the two LUTs can be disconnected by a configuration bit. BFG implements a soft S44 generator in `tiles::S44`, using the `tiles::LutB` generator for each 4-LUT. Since `tiles::LutB` implements a complete CLB by itself (a LUT with a register), we effectively create a *fracturable* S-44 CLB that can be split into two independent 4-LUT CLBs. The i_1 configuration bit is stored in another `atoms::Sky130Dfxtp` flip-flop, which is inserted into the scan chain between the two 4-LUTs.

The addition of dedicated carry-chain logic within CLBs is known to significantly improve the performance of arithmetic operations and is a common feature in modern fabrics [40], such as the UltraScale devices from Chapter 3. The `tiles::S44` generator includes the carry-chain logic from `atoms::Sky130Carry1` (Sec. 4.5). When configured as two seperate 4-LUTs, one LUT stores the propagate bit and the other stores the generate bit (P and G Fig. 4.5 respectively). These are input to the dedicated carry logic, which computes the sum and carry-out, before feeding the latter to the next bit in the chain as the carry-in (the next bit is an identical instance of the same structure). The sum value can then be output directly or registered within the CLB. Since the propagate and generate signals are retrieved from



(a) Recipe for the S-44 layout, from `tiles::S44`



(b) Generated layout of the soft S-44 CLB

each LUT in parallel for all bits in the sum, the delay of the overall addition operation over N bits scales with the fast-path sequence of carry computations through $N-1$ bits of them.

Figure 4.35a shows how the `S44` generator instantiates and physically arranges the other BFG generators to compose the soft S-44 cell. The top instance of `LutB` is generated with an additional input multiplexer at its S_2 input (Fig. 4.22), and becomes LUT i_2 in Figure 4.34. The other instance (i_0) of `LutB` is generated with the optional second bypass input X_2 enabled; the output sum from the carry logic will be connected here. Between the two LUTs a `RowGuide` is added which contains an instance of the carry chain bit `Sky130Carry1` and a flip-flop to store the fracture state of the soft S-44. Once these structures are aligned,

a wire is manually drawn to connect the carry-in and out wires to matched points at the vertical boundaries. When S-44 tiles are placed next to each other, the carry-out from one instance will connect to the carry-in of its neighbour by abutment. The router can be called to complete the remaining connections. Figure 4.35b shows the final layout. We note that this cell is incomplete, with a few of the described connections between instances remaining to be implemented. The reader is encouraged to consult the source, `tiles/s44.cc`, for detail.

4.7 $N : 1$ Interconnect Multiplexer

The `atoms::Sky130InterconnectMux1` class generates an $N : 1$ one-hot transmission gate multiplexer. Because this multiplexer is programmable, it is suitable for use in the interconnect topologies we saw in Chapter 3. Configuration is stored by flip-flops in a scan-chain and loaded by serial bitstream, as in the CLBs (Secs. 4.4 and 4.6); each bit of configuration memory controls one gate. This generator is an example of the third and most enlightened generator design pattern discussed in Section 4.1, where we overemphasise the separation of concerns within the layout, and is subsequently decomposed into several layers of sub-generators. We discuss these first: the `atoms::Sky130SimpleTransistor` in Section 4.7.1; the `atoms::Sky130TransmissionGate` in Section 4.7.2; and the `atoms::Sky130TransmissionGateStack` in Section 4.7.3, before describing how they are composed to build the overall multiplexer in Section 4.7.4.

4.7.1 Simple transistor

✓ DRC clean ✓ LVS pass

`atoms/sky130_simple_transistor.h`
`atoms/sky130_simple_transistor.cc`

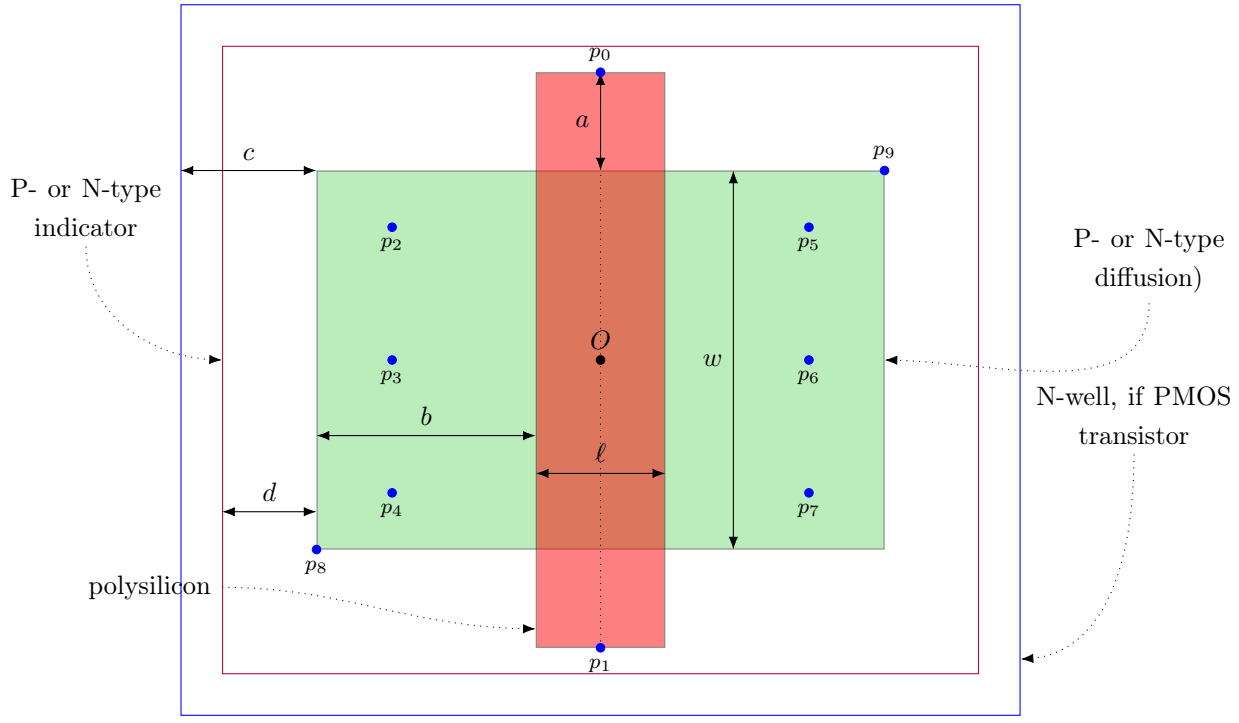


Figure 4.36: Template for the `Sky130SimpleTransistor` layout

The `atoms::Sky130SimpleTransistor` generates a cell containing a single transistor for the SkyWater 130 nm process. It provides a convenient abstraction to map common transistor choices to corresponding circuit and layout, as shown in Figure 4.36 (cf. Fig. 2.28). Consistent programmatic interfaces are provided to:

- choose between P- and NMOS transistors, their operating voltages and their various sub-types (high voltage threshold, regular, low voltage threshold);
- adjust the sizing of the gate in width (w) and length (ℓ);
- extend the polysilicon gate past the diffusion above or below (a);
- automatically size diffusion (b) to stack the transistor to the left, right or neither and
- generate a `Circuit` instance with the correct foundry transistor model.

Though it does not currently, the generator could also easily be extended to automatically add, size, and space (c, d) any necessary additional layers, such as P- or N-type diffusion indicators and P- and N-wells. In practice, this is not usually useful because the transistor is placed near others of its type, and only a single N-well needs to be drawn around all of them. The `geometry::Rectangle` class (Sec. 2.3) even provides helper functions to merge many such rectangles into a single large bounding box (`Rectangle::AccumulatedOver`), but in practice it is usually simpler for as a designer to nominate the boundaries themselves. However, for designs where positioning is less explicit, it would still be a useful feature.

The generator saves key points $p_i \mid i \in \{0, \ldots, 9\}$ in the design to allow the caller to quickly access commonly needed geometric points, without having to re-calculate sizing and spacing for each. These are:

- `poly_top_centre`, `poly_bottom_centre`: centre-top and centre-bottom of the polysilicon, typically for joining additional polysilicon (p_0, p_1);
- `via_`X`_diff_`Y for $X \in \{$`left`, `right`$\}$, $Y \in \{$`upper`, `middle`, `lower`$\}$, positions where vias (of the appropriate layer) can be placed (p_2–p_7); and
- `diff_lower_left`, `diff_upper_right`: defining corners of the diffusion rectangle (p_8, p_9).

Named points are the most broadly portable annotations in the framework, largely because they persist with the layout beyond the lifetime of the generating object. They can, however, be cumbersome, so the transmission-gate-stack generator provides more structured interfaces to retrieve via locations and polysilicon landmarks as well. The `Sky130SimpleTransistor::ViaPosition` and `Sky130SimpleTransistor::Locate` methods, shown in excerpt from the class header file in Listing 4.7, make it easier for designers to find operative coordinates in their code. Concretely, they don't have to assemble a string key before they can find anything, which also makes the interface more readable.

Listing 4.7: Structured access to layout properties post-parameterisation in `Sky130SimpleTransistor`

```
74  enum Landmark {
75    POLY_TOP_CENTRE,
76    POLY_BOTTOM_CENTRE
77  };
78  geometry::Point Locate(const Landmark &alignment);
79
80  enum ViaPosition {
81    LEFT_DIFF_UPPER,
82    LEFT_DIFF_MIDDLE,
83    LEFT_DIFF_LOWER,
84    POLY_UPPER,
85    POLY_MIDDLE,
86    POLY_LOWER,
87    RIGHT_DIFF_UPPER,
88    RIGHT_DIFF_MIDDLE,
89    RIGHT_DIFF_LOWER
90  };
91  // Calculates locations of key via positions for convenience.
92  geometry::Point ViaLocation(const ViaPosition &via_position) const;
```

The terminal names used by the underlying foundry transistor model are also abstracted. By convention these are resppectively `d`, `g`, `s`, `b` for the drain, gate, source and substrate. Instead of relying on convention and explicitly writing "`d`", for example, the user can query the generator with the `Sky130SimpleTransistor::TerminalPortName` and the `Terminal` enum[5]. That is,

```
std::cout << "Drain port: " << transistor_generator->TerminalPortName(
    Sky130SimpleTransistor::Terminal::DRAIN) << std::endl;
```

will print

```
Drain port: d
```

Finally, Figure 4.37 shows how circuit models generated by `Sky130SimpleTransistor` correspond to the generator's input parameters, after automatic conversion to SPICE (for brevity). Note in particular how the foundry model name and instance parameters change.

[5]This particular interface is a clunky C++-ism, but could easily be wrapped with a series of friendlier methods like `DrainPort`, `SourcePort`, and so on.

(A text-format protocol buffer.)

```
fet_type: NMOS
width_nm: 340
length_nm: 150
stacks_left: false
stacks_right: false
poly_overhang_top_nm: 0
poly_overhang_bottom_nm: 0
```

(a) Input parameters for regular NMOS

```
.SUBCKT Sky130SimpleTransistor
+ s d g b
* No parameters

xfet
+ d g s b
+ sky130_fd_pr__nfet_01v8
+ l='150n' w='340n'

.ENDS
```

(b) Generated circuit model for Fig. 4.37a

(A text-format protocol buffer.)

```
fet_type: PMOS_HVT
width_nm: 500
length_nm: 170
stacks_left: false
stacks_right: false
poly_overhang_top_nm: 0
poly_overhang_bottom_nm: 0
```

(c) Text-format parameters for high-V_T PMOS

```
.SUBCKT Sky130SimpleTransistor
+ s d g b
* No parameters

xfet
+ d g s b
+ sky130_fd_pr__pfet_01v8_hvt
+ l='170n' w='500n'

.ENDS
```

(d) Generated circuit model for Fig. 4.37c

Figure 4.37: Correspondence of two output `Sky130SimpleTransistor` circuit models to their input parameters

4.7.2 Transmission Gate

✓ DRC clean ✓ LVS mostly† pass

`atoms/sky130_transmission_gate.h`
`atoms/sky130_transmission_gate.cc`

The `atoms::Sky130TransmissionGate` generator uses two instances of a `atoms::Sky130SimpleTransistor` to lay out a transmission gate. Among the options it provides are:

- the minimum cell height;
- the width of the cell, as a function of:
  - whether the gate will be stacked against (abut another transmission gate) on the left or right; and
  - the expected polysilicon pitch in the parent cell;
- the position (above or below) of a polysilicon tab at each transistor where a via can be placed to connect to it;
- the minimum spacing of the transistors and their tabs from the vertical edges;
- the placement of tabs to align with a regular grid;
- the sizing of the underlying transistors, which it passes to the appropriate generator;
- whether the N-well should be drawn;
- whether the polysilicon tabs should avoid possible conflict with the nearest positions for vias to the diffusion of each transistor; and
- whether space should be left for a horizontal metal m_0 channel at the top and/or bottom of the cell.

Figure 4.38 shows three examples of layouts generated by the class with various input parameters. Note how the polysilicon tab positions (north or south) vary. When the `stacks_left` or `stacks_right` are true, the cell boundary is drawn at the respectively western or eastern diffusion edge, otherwise half of the appropriate spacing is inserted before the corresponding edge. Writing the transmission gate generator to accommodate the space of designs these options span is easier because the minutiae of sizing the transistors themselves is taken care of by the `Sky130SimpleTransistor` class.

†We note one caveat to LVS passing. `Sky130TransmissionGate` typically generates a circuit that pairwise connects the sources and drains of each transistor together, yet the corresponding metal wire between them is not drawn. This is convenient for the consuming `Sky130TransmissionGateStack` generator, which does draw the metal wires, because it simplifies assembling the circuit by one step. This is not strictly self-consistent — the remedy is

to add an option to actually draw the connecting wire or, for the pedantic, an option to split the output circuit to match the layout. We claim that it is *mostly* passing because the circuit and layout match with either remedy.

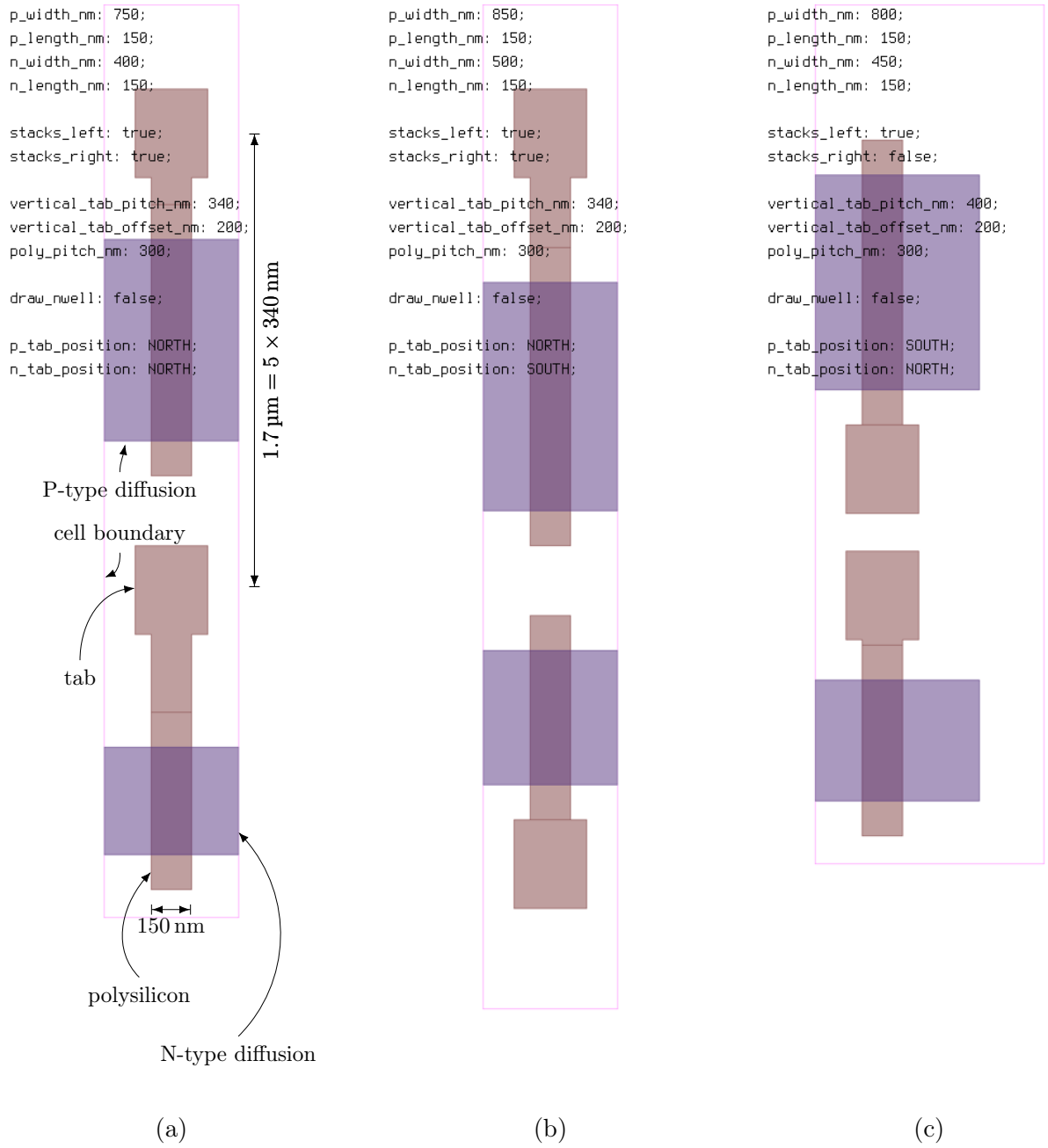


Figure 4.38: Example layouts generated by the `Sky130TransmissionGate` class with select parameters

4.7.3 Transmission Gate Stack

✓ DRC clean ✓ LVS pass

`atoms/sky130_transmission_gate_stack.h`
`atoms/sky130_transmission_gate_stack.cc`

The `Sky130TransmissionGateStack` class generates horizontally-stacked transmission gates. Its main parameter is a sequence of nets (signals) specifying the terminals of the stack. In fact, the parameter is a *list* of sequences: each inner sequence defines the terminals of a *contiguous* group of transmission gates, and the stacks for *all* sequences are themselves stacked next to each other with requisite spacing. The overall layout is a simple composition of `Sky130TransmissionGate` instances, as many as necessary for each sequence, stacked in a row by an instance of `RowGuide`.

As an example, consider the net sequence (A, B, C, D, E). The corresponding circuit, as would be generated by `Sky130TransmissionGateStack`, is shown in Figure 4.39. Terminal names are allocated successively from the sequence to drain/source and control nodes, and since intermediate source/drain nodes are shared, an N_T-terminal sequence requires

$$N_G = \left\lfloor \frac{N_T - 1}{2} \right\rfloor$$

transmission gate instances. The layout generated for this sequence, including the component planning objects, is shown in Figure 4.39.

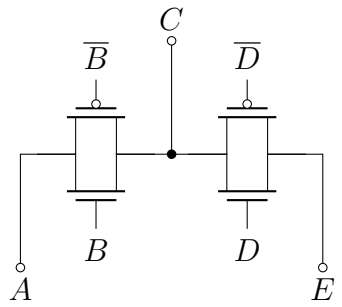


Figure 4.39: `Sky130TransmissionGateStack` circuit for the sequence (A, B, C, D, E)

After placing each of the child instances, `Sky130TransmissionGateStack` connects corresponding transistors from drain to drain (or source to source) over the first wire layer, metal m_0. Because each child transistor records the positions of vias to the diffusion regions, all we need to do is query them and determine the most suitable for the configuration, and then use a `PolyLine` to create an appropriate wire using PDK (foundry) rules. Since the stack generator collects many P- and NMOS transistors together, it draws the final N-well and N-/P-type indicator regions (as shown in the figure) around each group. Likewise for other tidy-ups: the special SkyWater 130 nm `npc.drawing` layer, which we have avoided describing

until now due to its overbearingly bureaucratic role, is drawn around groups of polysilicon contacts to indicate via connectivity.

The stack generator passes many parameters directly through to the underlying components. For example, the P- and NMOS transistor widths and lengths ($w_{p/n}$ and $\ell_{p/n}$) are specified to the transmission gate generator, which in turn specifies them to the transistor generator. Consequently, most of the parameters listed for `Sky130TransmissionGate` are exposed here also. (The reader is encouraged to read the source for the full list, `atoms/sky130_transmission_gate_stack.h`.) Figure **??** provides additional examples of output layouts along with their input parametesr, and shows how multiple net sequences manifest in layout.

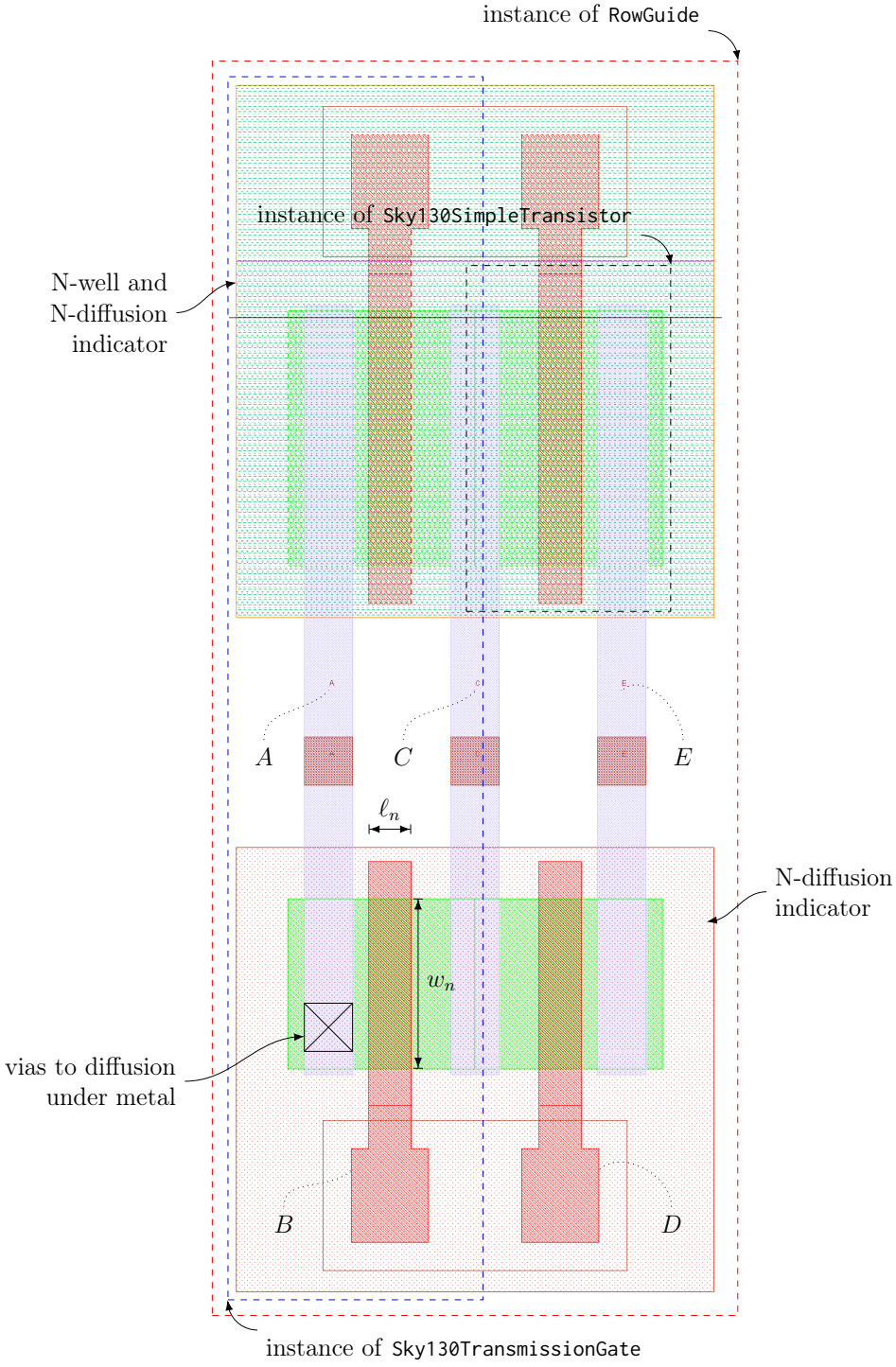
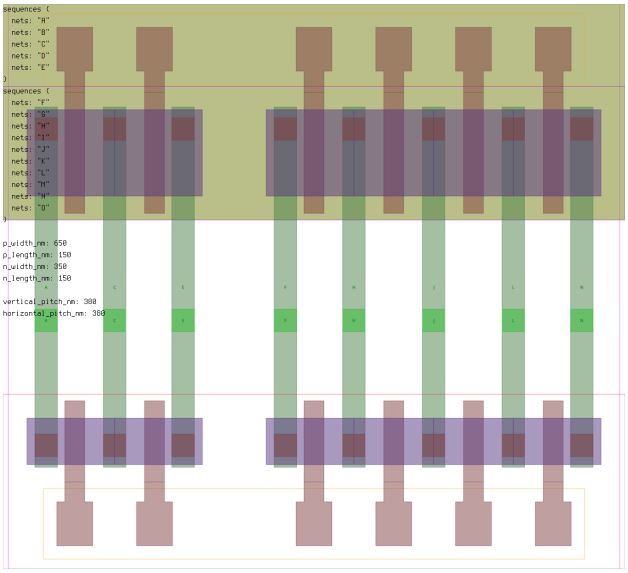
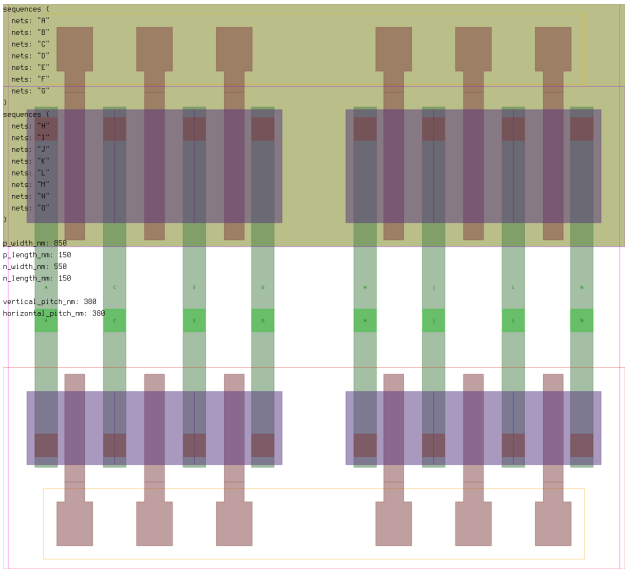


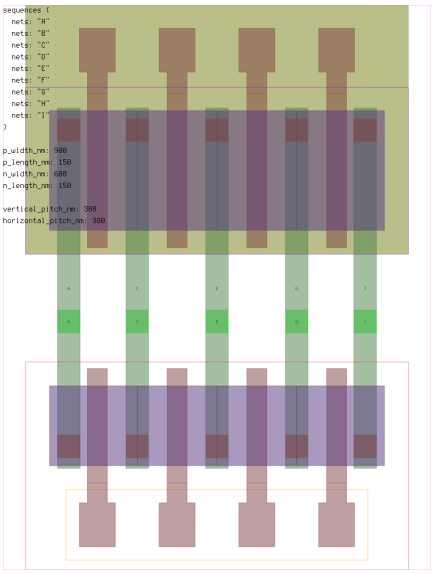
Figure 4.40: Layout of a 5-terminal sequence of transmission gates, generated by `Sky130TransmissionGateStack`



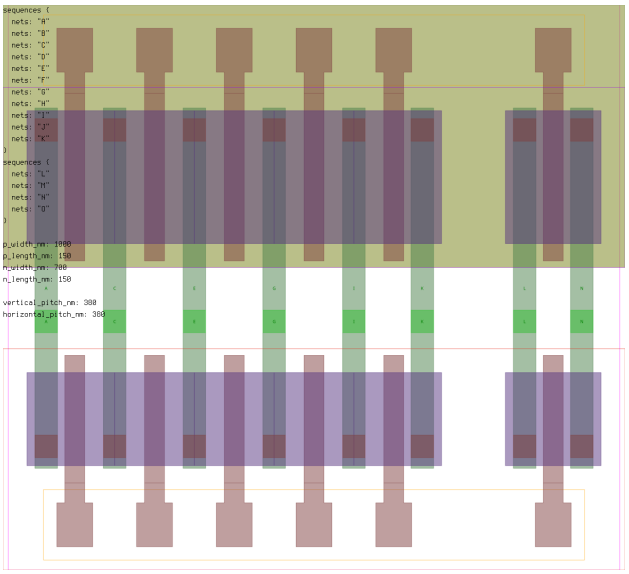
(a) Net sequences: $(A, \ldots, E)$, $(F, \ldots, N)$



(b) Net sequences: $(A, \ldots, G)$, $(H, \ldots, N)$



(c) Net sequence: $(A, \ldots, I)$



(d) Net sequences: $(A, \ldots, G)$, $(H, \ldots, N)$

Figure 4.41: Transmission-gate-stack layouts with varying transistor sizes and net sequences
(Note that the O input never completes an additional gate, so is always omitted)

4.7.4 Composition

✓ DRC clean ✓ LVS pass

`atoms/sky130_interconnect_mux1.h`
`atoms/sky130_interconnect_mux1.cc`

Finally, we turn our attention back to the N:1 one-hot transmission-gate multiplexer intended for use in the fabric interconnect. The primary parameter for BFG's `Sky130InterconnectMux1` is the number of inputs it takes N; the overall circuit is shown as a function of N in Figure 4.42. The design is based around a one-hot transmission-gate multiplexer; each input $x_i \mid i \in \{0, \ldots, N-1\}$ connects to a transmission gate g_i, which is in turn controlled by complementary signals from a dedicated register FF_i. The registers are arranged in a scan-chain as they are in the CLB (sorry John). When gate i is turned on, input x_i is connected and drives the output Z through an inline buffer. We opt for a one-hot scheme because it would have cost equal or more area to build a decoder circuit, for $N < 8$. The configuration bitstream is responsible for enforcing one-hot encoding; nothing else prevents multiple inputs being connected simultaneously.

The layout recipe is shown in Figure 4.43. We use a `MemoryBank` to manage the placement of every instance in a single collection of rows. An instance of the `Sky130TransmissionGateStack` implements the N:1 one-hot multiplexer in the central row. It is made tall enough to accommodate all N inputs being drawn in as a single-layer bus from the left (that is, at least $(N+1)$ wire pitches tall). The output buffer is then added to its right. Note that BFG's abstractions allow us to mix and match rows of any height, whereas this would not be possible with generic standard-cell placement tools — otherwise the design does resemble a simple standard-cell floorplan.

A key feature of the layout is the addition of dedicated wire routing channels in a column to the left and an additional row above. This is because SkyWater 130 nm only provides two layers for most logic routing. Of the five (total) metal wiring layers, the first (m_0) is high-resistance and supports limited wire lengths[6]. Those at or above m_3 have large minimum-widths and are intended to span large distances, so are reserved for clock and power distribution. This leaves only m_1 and m_2. To make to the multiplexer easier to route when tiled, we set aside these dedicated horizontal and vertical channels. Though they could automatically be sized for the number of inputs, they are currently simply exposed as a parameter (which by default supports 8 tracks).

The also generator has an experimental "inside out" setting, which toggles the rotation of every row. Doing so changes which rail is at which vertical boundary: power or ground at the top or bottom. This can make it easier to tile the interconnect multiplexer with other cells, *without* changing the orientation of input/output wiring.

The layout proceeds as follows:

1. The N memories (instances of `Sky130Dfxtp`) are split into two approximately even groups, one of which is allocated to the bottom half of the cell and the other to the

[6]This is because it isn't really "metal" but a rather "local interconnect" layer (Sec. 1.7.5.1).

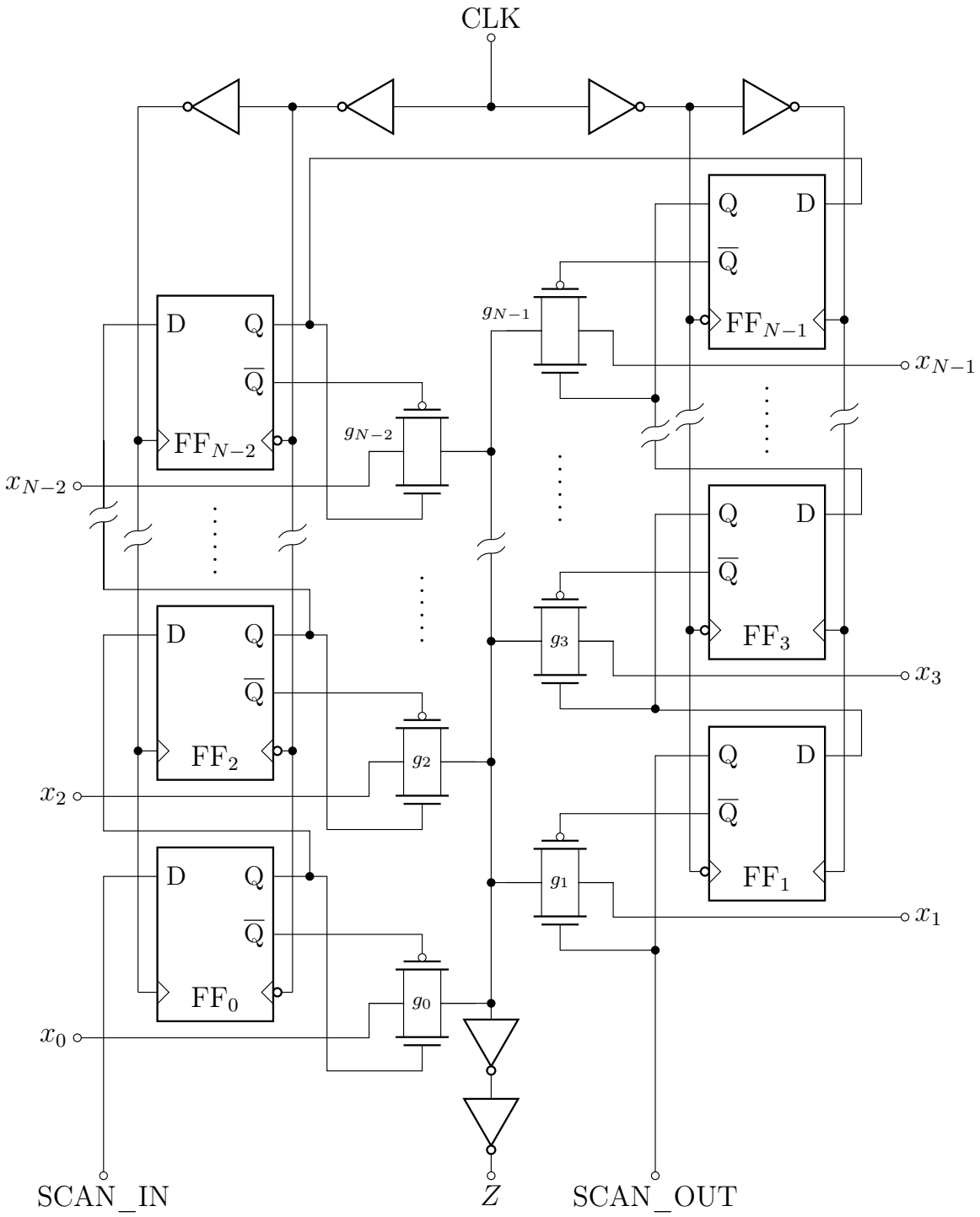


Figure 4.42: N:1 Interconnect Multiplexer Circuit

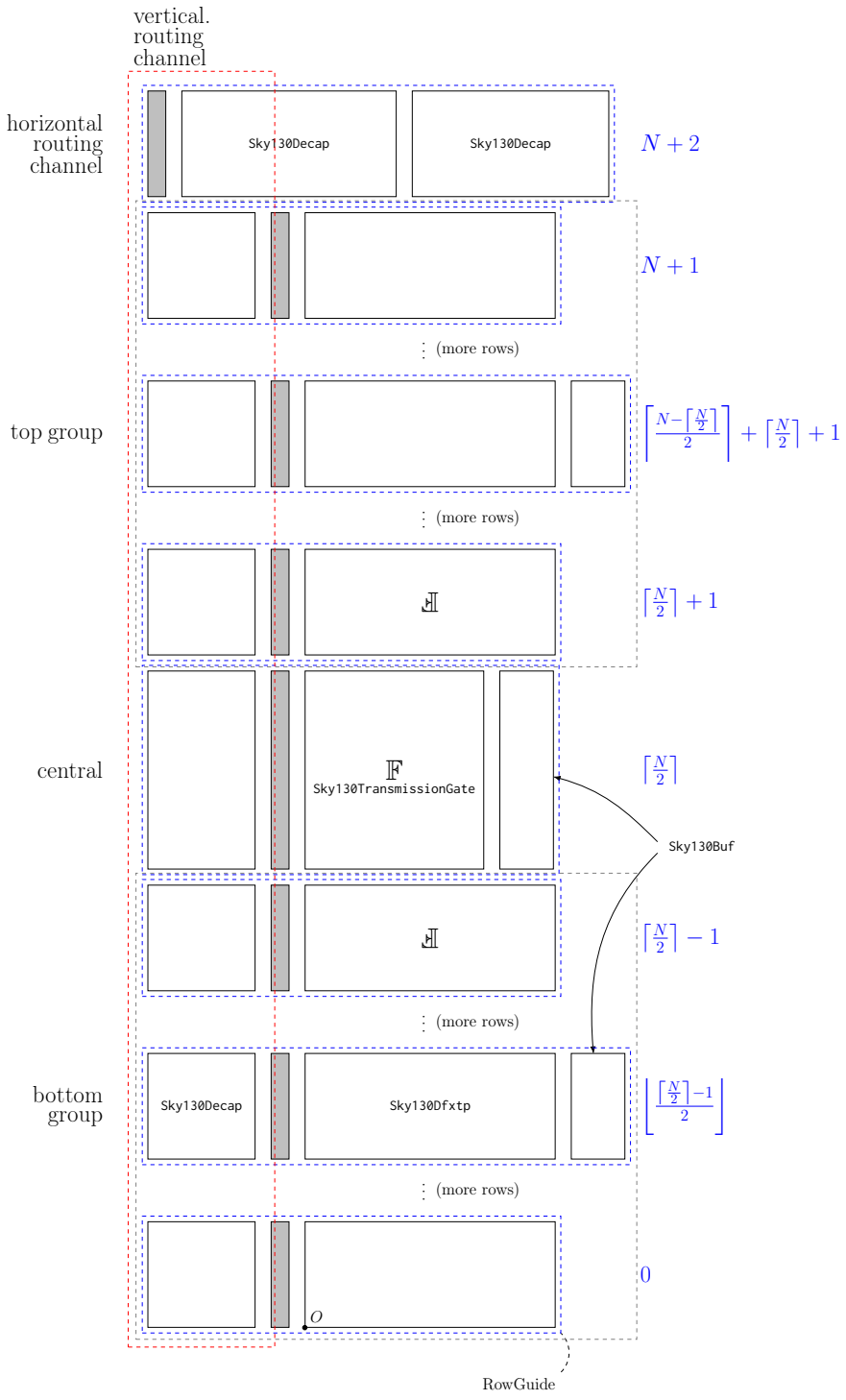


Figure 4.43: Layout recipe for the $N:1$ transmission-gate multiplexer

top. The flip-flops have their individual input clock buffers disabled. The optional $\overline{Q}$ port is added to the flip-flops as well: each transmission gate needs complementary control signals which we can then source directly from the memory.

2. An instance of `Sky130Buf` is added to the central row *of each group* to generate the local clock signal and its complement.

3. The "central" row is allocated an instance of the `Sky130TransmissionGateStack` configured for N inputs. Because there is only one output, the transmission-gates can only be stacked in pairs or not at all. Accordingly, the list of net sequences for the generator becomes:

$$\begin{aligned} \{(x_0, S_0, Z, S_1, x_1), (x_2, S_2, Z, S_3, x_3), \ldots, (x_{N_2}, S_{N-2}, Z, S_{N-1}, x_{N-1})\} & \quad \text{for even } N \\ \{(x_0, S_0, Z, S_1, x_1), (x_2, S_2, Z, S_3, x_3), \ldots (x_{N_1}, S_{N-1}, Z)\} & \quad \text{for odd } N \end{aligned}$$

where $x_i \mid i \in \{0, \ldots, N-1\}$ are the inputs, S_i is the control wire enabling input i, and Z is the output.

4. Rows alternate between 0° and 180° rotation. Here, care is taken to ensure that the central row has a fixed orientation independent of N so that a memory r rows away from the center always has its inputs and outputs on the same side. This greatly simplifies the routing problem. To achieve this, the `MemoryBank` is simply configured to rotate the first row according to whether N is even or odd.

5. An instance of `Sky130Buf` is appended to the central row, next to the transmission gates.

6. The vertical routing channels are created by placing a sized instance of `Sky130Decap` at the left of each row. As noted, these have a minimum size defined by a class parameter. The designer can also specify a horizontal pitch for the overall cell; the routing channels are sized to meet the minimum width but also restore the overall cell width to an integral multiple of the pitch, as shown in the excerpt of Listing 4.8.

Listing 4.8: Size routing column width to fit overall cell with to pitch, from `atoms/sky130_interconnect_mux1.cc`

```
556   // With the bottom memories and their clock buffers placed, we have the
557   // minimum width of the cell and so can size the left routing channel and
558   // output row to make the overall mux meet the pitch requirement:
559   uint64_t fixed_row_width = bank.Row(num_ff_rows_bottom - 1).Width();
560   uint64_t horizontal_pitch_nm = parameters_.horizontal_pitch_nm.value_or(
561       Parameters::kHorizontalTilingUnitNm);
562   uint64_t vertical_channel_width_nm = Utility::NextMultiple(
563       parameters_.vertical_routing_channel_width_nm.value_or(1380) +
564           fixed_row_width,
565       horizontal_pitch_nm) - fixed_row_width;
```

7. The horizontal routing channel is similarly created by placing sized instances of `Sky130Decap` in a new row at the top of the layout. If N is even, then this additional row will restore the total number of rows N_R to an even number, which means that the cell can be tiled vertically without rotation; if N is odd, then N_R was already even before this step, and we have to add an even number of horizontal routing rows to achieve the same outcome.

8. Additional `Sky130Decap` cells are generated to fill the gaps in each row on the right side of the layout. For the cells listed here, these gaps occur under the input clock buffers in each group of memories, and in the central row next to the output buffers.

9. Control wires from the output of each memory (both Q and $\overline{Q}$) are routed to the transmission gates. The polysilicon pitch of the gate stack is calculated to accommodate two m_2 wires laid vertically over the top of each gate, so long as successive pairs of wires travel in opposite directions. Subsequently, the bottom memories connect to the even numbered gates and the top memories connect to the odd numbered gates. Because there is only one memory per row, we only need to ensure that the output port on the memory is on a unique horizontal m_1 wiring track to guarantee connectivity. If there are multiple output ports, the one closest to the central row is chosen. This mechanism breaks down for sufficiently large N: at some point, enough memories are stacked vertically that there remains no way to route them all to the transmission gates. The limit on N depends on the the transmission gate pitch and on the width of the memories themselves: using a 500 nm pitch and the `Sky130Dfxtp` memory cells, it works out to be $N = 7$.

10. The scan chain is connected. On each memory, we re-use the output port connected to the control wiring. The input port is chosen to be the furthest from the central row in the hope that this puts the input and output on different tracks. The vertical control wires laid in (9) present a large contiguous set of shapes that we must avoid. The simplest mechanism way to do this is to pick columns to the left or right of that block. We define *which* columns are used statically (i.e., the first on the left and the first on the right), but the exact (x-)position of the columns is determined by tracking the bounds of the control wiring as it is laid.

11. The transmission gates are connected to the output buffer, and the output buffer is connected to the output port.

12. The inputs are drawn out from the gate stack into to the vertical routing channel, where they are made to land on a regular grid (whose pitch and offset are generator parameters) but staggered to ensure that they are mutually connectable.

13. Power, clock, and ground straps are laid vertically on m_2 using statically-defined vertical column indices (just as in (10)). Then, these are automatically connected

to ports using horizontal m_1 wires as necessary (the `Layout::MakeVerticalSpineWithFingers` method makes it convenient to do this.) The static column indices are relative to either the central control wiring or the cell's east edge. The adaptability this yields ensures all of the wiring structure remain correct by construction for $3 \leq N \leq 7$.

The layout of an example $6:1$ multiplexer ($N = 6$) built by `Sky130InterconnectMux1` is shown in Figure 4.44. Examples for $N \in \{3, 4, 5, 7\}$, shown in Figure 4.45, show how the design steps adapt to various sizes.

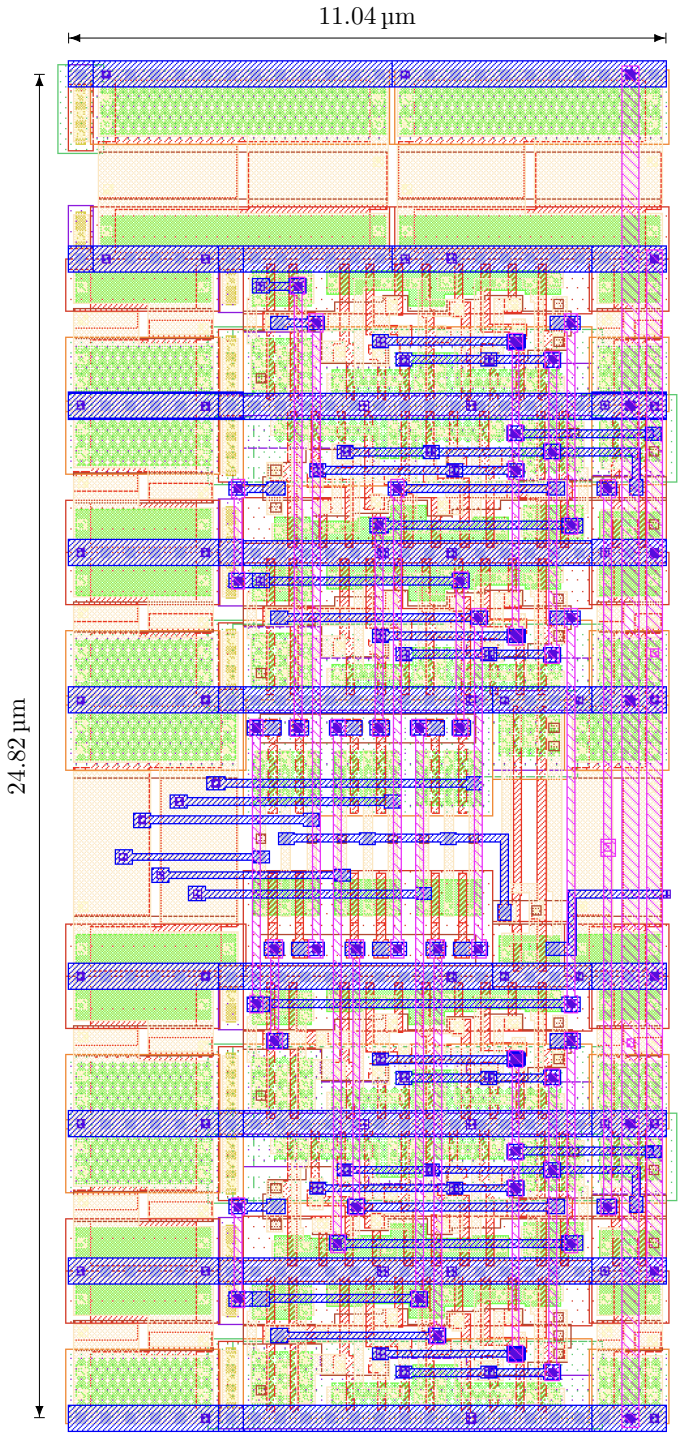
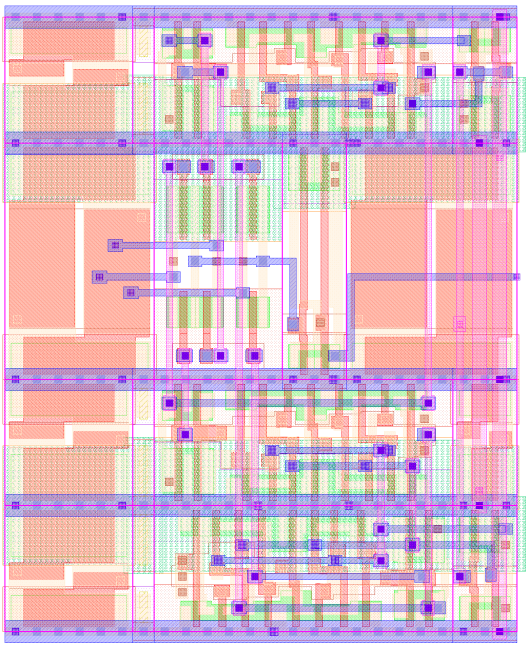
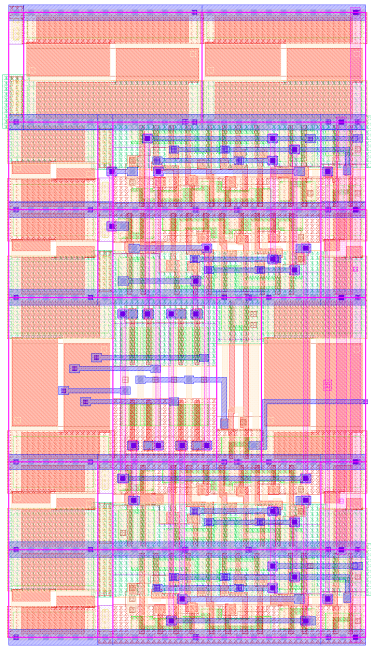


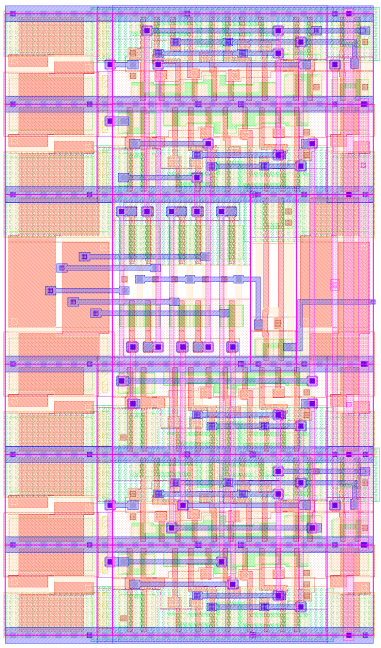
Figure 4.44: A 6 : 1 one-hot transmission gate multiplexer generated by `atoms::Sky130-InterconnectMux1`



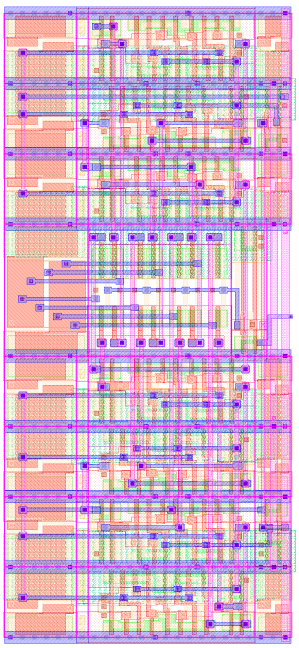
(a) 3 : 1



(b) 4 : 1



(c) 5 : 1



(d) 7 : 1

Figure 4.45: $N : 1$ multiplexers generates by `Sky130InterconnectMux1`

4.8 $M : 2$ Interconnect Multiplexer

✓ DRC mostly clean ⚠ LVS untested

`atoms/sky130_interconnect_mux2.h`
`atoms/sky130_interconnect_mux2.cc`

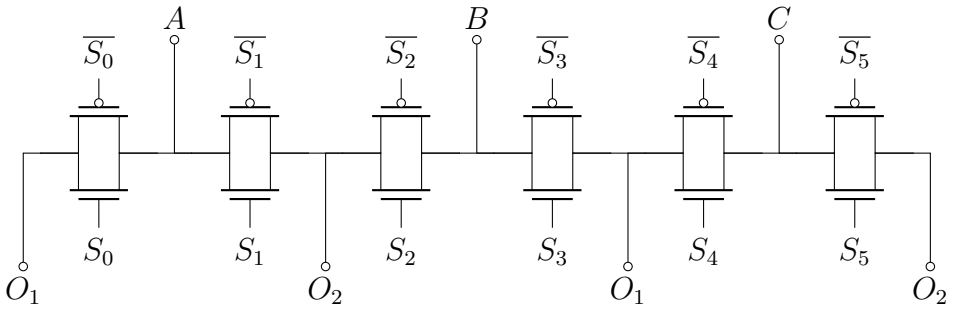


Figure 4.46: M:2 interconnect multiplexer circuit
Registers and output buffers are not shown, but are analogous to Fig. 4.42.

The `atoms::Sky130InterconnectMux2` generates a single cell that contains two *shared* $N : 1$ one-hot transmission-gate multiplexers. The two N:1 "logical" multiplexers have $N-1$ inputs in common, so the overall structure has $M = 2N - (N - 1) = N + 1$ inputs, and 2 outputs. We call it an $M : 2$ multiplexer. The generator is a direct C++ sub-class, in the software polymorphism sense, of the $N : 1$ generator from the previous section. It composes the same base generators and uses the same layout strategy as in the single-output case, with the following exceptions:

1. The transmission gate stack is configured to alternate between the two outputs. Figure 4.46 shows the transmission-gate circuit, omitting the registers used to configure each gate, their scan-chain, and the output inverters for each of O_1 and O_2, since these follow from the design of the $N : 1$ case in Figure 4.42. The structure implements exactly the compact transmission-gate layout for shared multiplexers described in Section 3.2.1.

2. There are now two independent outputs, so there are two output buffers.

3. Configuration memories are now stored two per row to balance the aspect ratio of the layout. The bottom registers control one logical half and the top registers control the other.

4. The transmission gate stack is centred in its row to align with the boundary between the two memories in each row above and below it. This allows the horizontal tracks from the outputs of memories in the each row to travel toward the stack without colliding.

5. Routing control signals is significantly more complex.

The $N:1$ multiplexer generator (Sec. 4.7) could easily construct a correct allocation of wires by splitting signals to the north and to the south; there was no contention for wiring between registers. In the $M:2$ case generator this is no longer true: identical memory cells in the same row place their output ports on the same track. Although their respective generators offer multiple output ports to choose from, we avoid this to save horizontal tracks for scan-chain and clock signal wiring. Instead, we employ a brute-force search to find an allocation of transmission-gates to memories that causes no conflicts, proceeding as follows:

1. We iterate over the rows of memories and allocate each memory to the nearest available transmission gate. Once a gate is allocated it is removed from consideration.

2. We check whether the horizontal wiring in each row would cross under the allocation, by projecting the source and destinations of each wire path onto the same line and then checking for intersection.

3. If there is no conflict, we are done. If there is a conflict, we check the next permutation of row ordering until there are none left.

This approach could be replaced with the automated router, which would be easier to use but is 10–30× slower. It is instead an example of the kind of partially-automated strategies that can be employed when there are only a handful of constraints.

Figures 4.47 and 4.48 respectively show examples of $5:2$ and $7:2$ multiplexer layouts generated by `Sky130InterconnectMux2`. A single $6:1$ (Fig. 4.44) has a tiling area of 274.01 µm. Comparing this to the $7:2$ multiplexer, which has a tiling area of 436.83 µm, the shared $7:2$ structure saves 20.3% area over two individual $6:1$ structures. This is a significant but surprisingly small saving given the speculation of Chapter 3. If we remove the dedicated wiring channels added for SkyWater 130 nm, their areas are 171.37 µm and 311.82 µm respectively, and the total area saving by the shared multiplexer structure is only 9.02%. However, the number of configuration bits does not change between two shared and two independent multiplexers, and BFG's designs are clearly dominated by configuration memory. If a more compact storage memory was used — say, an SRAM or a latch — the benefits would be more pronounced. That said, there is still significant benefit to using the shared structure because it dramatically reduces wiring congestion. The number of input wires to a *shared* $N:1$ multiplexer is $N-1$ less than to two individual $N:1$ structures.

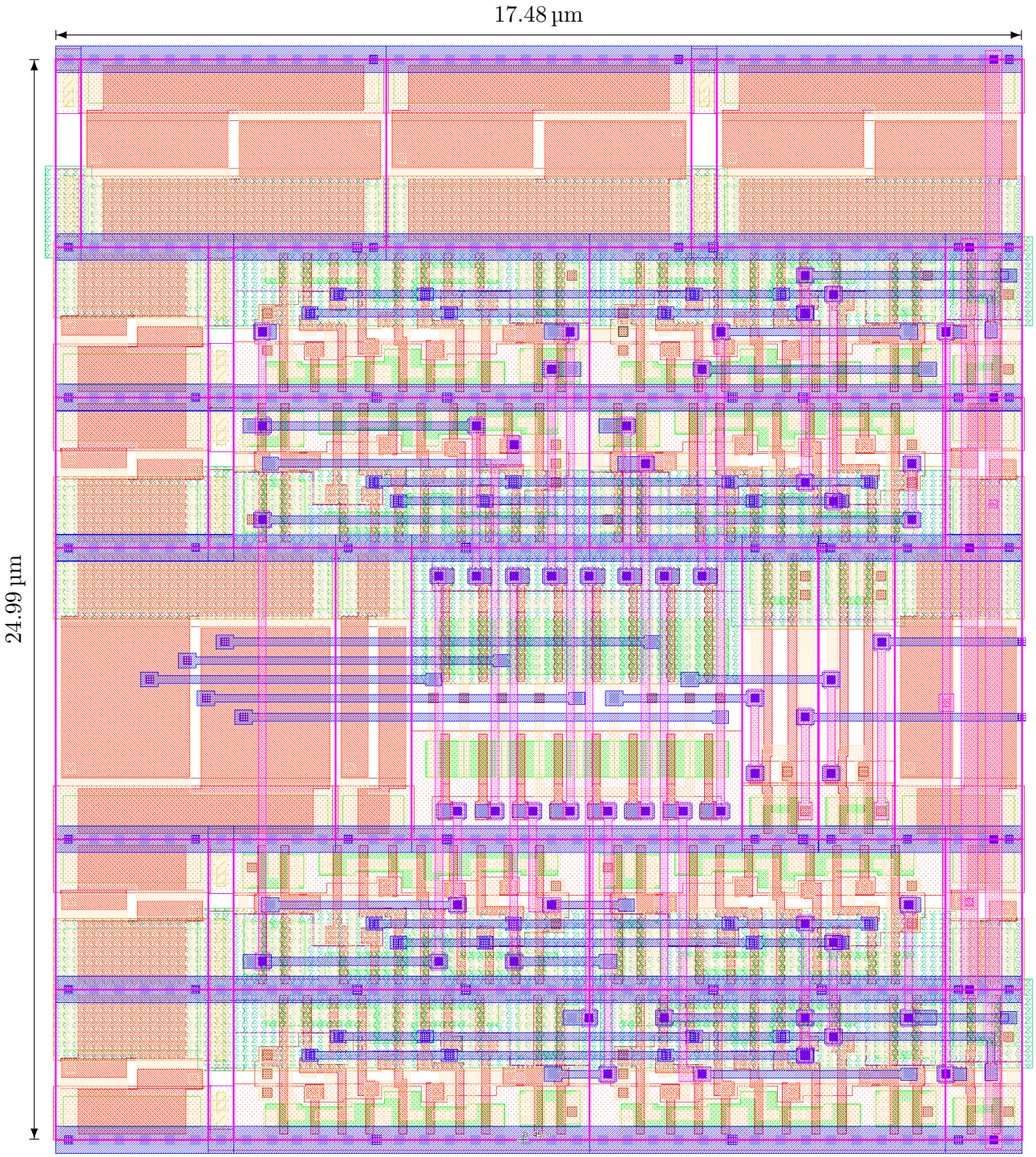


Figure 4.47: 5 : 2 Multiplexer layout generated by `Sky130InterconnectMux2`



Figure 4.48: 7 : 2 Multiplexer layout generated by `Sky130InterconnectMux2`

4.9 Interconnect Wire Block

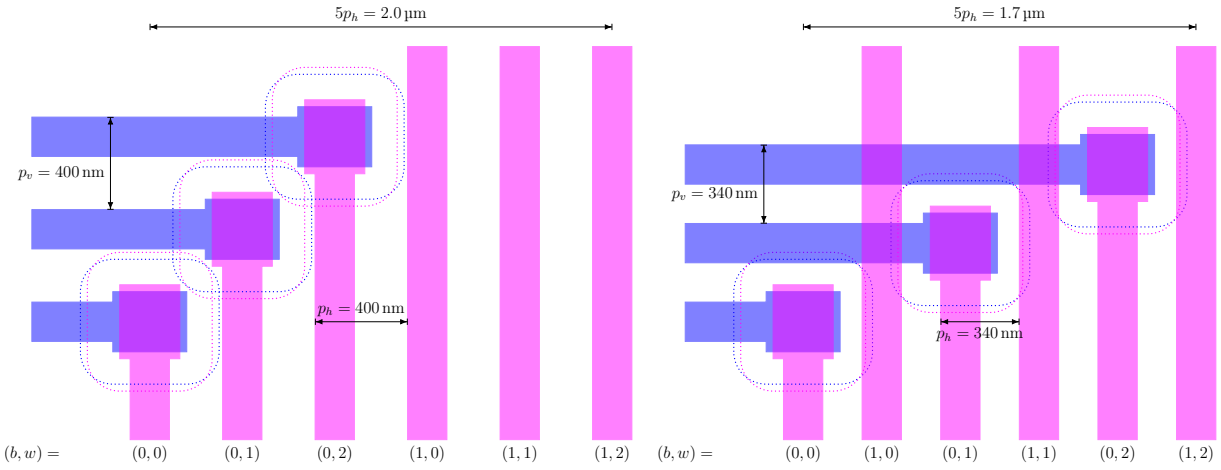
The `tiles::InterconnectWireBlock` generates layouts for collections of wires travelling together horizontally or vertically. A wire *bus* typically contains several wires that are laid out and terminated together. It is common in interconnect wiring (Ch. 3) to group buses by direction and length; we call these *bundles*. In the abstract, the criteria for what makes a bundle is not well-defined: it is up to the designer to use the grouping tiers effectively (or not at all). For example, all of the wires travelling north and spanning 12 tiles can be considered to be the same kind of bundle. These bundles travel together but must be independently selected for termination; in a given tile, one bundle will terminate, another will originate, and still others will simply pass through.

Accordingly, `tiles::InterconnectWireBlock` is organised into *channels*, *bundles* and *wires*. A bundle b is a collection of $N_{W,b}$ wires; channels c collect together $N_{B,c}$ bundles; and the block itself hosts N_C channels. By default all bundles pass straight through the block, with ports only at opposite ends. The designer can nominate which bundles, if any, are "broken out": the bundle is broken into two sections, and orthogonal wires are drawn from the lateral edges of the cell to the wires to connect to each section. These *taps* expose ports for sources and sinks of the wires themselves. By convention one section is referred to as the *input* and the other as the *output*. The designer can also opt to delete one of the two sections; in an interconnect it is common for a terminating bundle to restart in the same tile, but at the edges of the fabric there is no need to restart a new run. Broken-out bundles are nominated by their index within their channel.

The generator takes the following parameters:

- N_C, $N_{B,c}$, $N_{W,b}$, as described;
- the break-out indices and any deleted sections;
- details about the broken-out taps:
  - their position (offset) along the side of the tile; and
  - which side of the tile should be used;
- the wire width, pitch, and offset; and
- a choice of "conservative" or "modestly clever" layout mode.

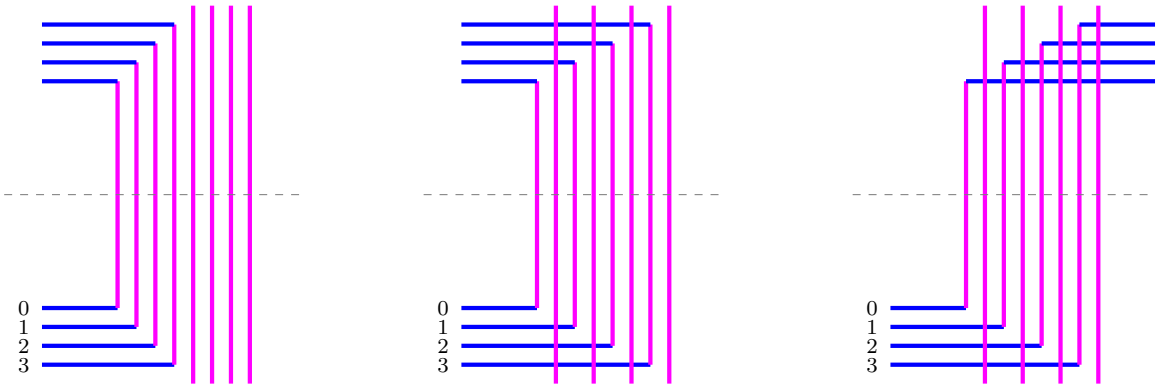
The layout mode defines how wires within a bundle are interleaved. The first regime is *conservative* because it avoids complexity: it lays out all of the wires of a single bundle next to each other. This is illustrated in Figure 4.49a. The disadvantage of this approach is that the vias connecting orthogonal break-out wires require mutual separation, and so the wires must be spread apart. The second regime is *modestly clever*: it interleaves the wires of *all* of the bundles within a single channel, as shown in Figure 4.49b, and thereby avoids adjacent vias. The wire pitch can subsequently be reduced for a more compact layout. In SkyWater



(a) Conservative (b) Modestly clever

Figure 4.49: Bundle layout strategies in the `InterconectWireBlock`

130 nm this results in a 15% reduction in cell width. There are two disadvantages, the first of which is additional complexity. We must ensure that adjacent bundles are never broken out and that there are at least two bundles in each channel.



(a) Conservative (b) Modestly clever (c) No problem

Figure 4.50: Skew between longest and shortest paths in `InterconnectWireBlock` layout regimes

The second disadvantage to the modestly clever approach is that the difference in length between longest and shortest wire within a bundle, ΔL, and hence the signal *skew*, now grow with the number of bundles in a channel. Consider the conservative layout of N_B bundles, each of N_W wires, within a single channel, as shown in Figure 4.50a. If the source and sink breakouts for a bundle are on the same side of a contiguous group of wire blocks, the

difference between the shortest (0 in the example) and longest (3) wires is

$$\Delta L = (N_W - 1)P_h + 2(N_W - 1)P_v$$

where P_h and P_v are the horizontal and vertical pitches[7], respectively. Note that this is independent of N_B. If instead we lay the wires out in the modestly clever regime (Fig. 4.50b), this quantity becomes:

$$\Delta L' = N_B(N_W - 1)P'_h + 2(N_W - 1)P'_v$$

The vertical component is unchanged, but the horizontal component now grows with N_B. The problem is mitigated by the fact that achievable pitches in the modestly clever approach are smaller $P'_{\{h,v\}} \leq P_{\{h,v\}}$, but the overall effect on skew must now be taken into account by the designer.

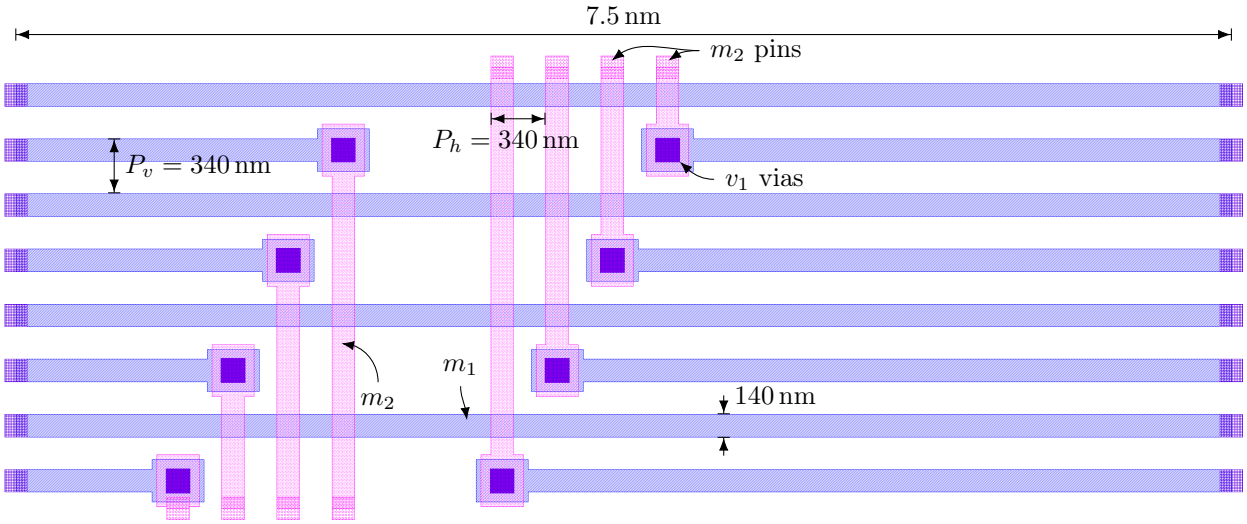


Figure 4.51: An interconnect wire block laid out by `InterconnectWireBlock`

Figure 4.51 shows an example `InterconnectWireBlock`-generated layout with $N_C = 1$, $N_{B,0} = 2$ and $N_{W,0} = 4$ under the modestly clever strategy; bundle 0 is tapped on alternate sides at 1 µm and 3 µm. The full input configuration file is as follows:

```
direction: TRACK_HORIZONTAL
layout_mode: MODESTLY_CLEVER
horizontal_layer: "met1.drawing"
via_layer: "via1.drawing"
vertical_layer: "met2.drawing"
```

[7]Note that "horizontal pitch" means "pitch in the horizontal dimension", *note the pitch of the horizontal wires*. That is the vertical pitch. Obviously I am telling you this because I keep getting them confused.

```
horizontal_wire_width_nm: 140
horizontal_wire_pitch_nm: 340
horizontal_wire_offset_nm: 170
vertical_wire_width_nm: 140
vertical_wire_pitch_nm: 340
vertical_wire_offset_nm: 170
length: 7500                # Internal units; 7.5 µm
channels {
  name: "6X"                # Names are used for the generated nets.
  bundles {                 # First bundle is broken out down and up.
    num_wires: 4
    tap: true
    break_out {
      alternate_side: false
      offset: 1000          # Internal units; 1.0 µm
    }
    break_in {
      alternate_side: true
      offset: 3000          # Internal units; 3.0 µm
    }
  }
  bundles {                 # Second bundle passes straight through.
    num_wires: 4
    tap: false
  }
}
```

4.10 Unnamed Columnar Architecture

In this section we will show, finally, how BFG (and silicon compilers like it) might be used to assemble a whole, modern FPGA fabric. Using the BFG framework (Ch. 2), the interconnect topology generator (Ch. 3) and the collection of parametrised circuit and layout generators previously discussed in this chapter, we will design and lay out a tile for our own UltraScale-like fabric. We will refer to the architecture as the Unnamed Columnar Architecture (UCA) when convenient.

We start with the overwhelming physical constraints. UltraScale and UltraScale+ devices are manufactured on 20–16 nm processes, whereas we initially targeted the SkyWater 130 nm process. As discussed in Section 4.7, we are limited to two usable wiring layers m_1, m_2 for most interconnect because lower layers are too high-resistance and upper layers are reserved for clock and power distribution. Using these layers also generally precludes using the transistor area beneath them for useful logic. Wire area is therefore expensive, and we must first size the interconnect to balance the its area usage.

Because interconnect wiring in UltraScale-like architectures is point-to-point, and each link spans L-many tiles, L-many bundles of wires must run in parallel to serve successive source/destination tile pairs. One bundle connects to a given tile, and the other $L-1$ must pass by to connect neighbours. Let N_B be the number of wires in each of these bundles. As described in Section 4.9, we collect same-length bundles into *channels*. Because wires are directional and we connect tiles with reciprocal links either vertically (north-south) or horizontally (east-west), each bundle is part of a reciprocal pair. The total number of wires $N_W^{(C)}$ that must be run in parallel to accommodate a channel is then

$$N_W^{(C)} = 2LN_W$$

Next consider that, in UltraScale-like architectures, interconnect wiring of all but the longest-length channels are duplicated for each of the two slices within a single tile. On the other hand, the dedicated wiring trunk between neighbouring tiles can exclude the 1-unit wire spans, since those can likely connect to neighbouring tiles directly at many points along the tile periphery. As shown in Chapter 3, UltraScale uses byte-sized bundles $N_W = 8$ with channel lengths $L \in \{1, 2, 4, 12\}$, so the total number of wires running north to south in an interconnect trunk would be

$$\begin{aligned} N_W^{(NS)} &= 2 \cdot 12 \cdot 8 + 2 \sum_{L \in \{2,4\}} 2 \cdot 8 \cdot L \\ &= 384 \end{aligned} \tag{4.1}$$

At the minimum SkyWater 130 nm wire pitch (340 nm) this would require a channel 130.56 µm wide. To appreciate the scale of such a monstrosity, consider that the S-44 CLB (Sec. 4.6) — the largest layout we seen thus far — is only 28.56 µm wide and 54.40 µm tall.

In order to reduce the wiring burden, we make a series of compromises. We first reduce the size of each bundle from a byte down to a nibble, $N_W = 4$. It is natural to ship bytes

together since they are a common unit of compute, so we choose to use nibbles (which can be composed to form a byte) instead. Next, we reduce the maximum span of a channel from 12 to 6 tiles. This also accounts for the overall larger size of each tile in a 130 nm process over a 16 nm one: a 6-tile wire is much longer in our device than a 6-tile wire in the UltraScale. Now, consider that the UltraScale's progression for channel lengths L is *almost* exponential base-2,

$$\begin{aligned} L &\in \{1, 2, 4, 12\} \\ &\in \left\{2^0, 2^1, 2^2, \frac{3}{4} \cdot 2^3\right\} \end{aligned}$$

This strongly suggests a deliberate binary partitioning of the the horizontal and vertical dimensions, where any length ℓ can be spanned in $\lceil \log_2 \ell \rceil$ wire lengths. The 25% reduction of the longest span, from 2^3 to 12, is probably a compromise between the partitioning property and the delay of propagating a signal along the wire, which we known to be quadratic in L. Accordingly, we remove the $L = 4$ channel and are left with

$$\begin{aligned} L &\in \{1, 2, 6\} \\ &\in \left\{2^0, 2^1, \frac{3}{2} \cdot 2^2\right\} \end{aligned}$$

With $N_W = 4$ and $L \in \{1, 2, 6\}$ the total number of wires in our interconnect trunk (Eq. (4.1)) reduces to

$$\begin{aligned} N_W^{(NS)} &= 2 \cdot 6 \cdot 4 + 2 \cdot 2 \cdot 4 \cdot 2 \\ &= 80 \end{aligned}$$

At our nominal wire pitch, this number of wires only requires 27.20 µm in width.

In the UltraScale, the size of each bundle N_W seems to correspond to the number of BLEs (CLBs) placed in any tile. Each has multiple outputs and can even be fractured so the mapping is not 1:1, but nevertheless they are organised such that units of N_W-many signals can be routed together to and from various nodes. We emulate this pattern, placing $N_W = 4$ CLBs at the heart of each slice. We use S-44 CLBs for their proven logic density, per Section 4.6.

We largely use the same configurations of multiplexers as in the UltraScale, since we do not yet have evidence for changing them. We scale the numbers of each for our smaller interconnect wiring and our reduced numbers of CLB inputs and outputs. The number of Input Interconnect Block (IIB) Stage 2 (S2) multiplexers, for example, is equal to the number of CLB inputs: $8 \times (6 + 2) = 64$ in UltraScale, which has eight 6-LUTs that have 2 bypass inputs each, and $8 \times (4 + 1) = 40$ per slice in our case. In the Output Interconnect Block (OIB) Stage 2 (S2), we reduce the number of multiplexers from the 96 logical per slice UltraScale to 48 logical per slice, scaling down linearly with our lower number of CLB outputs (16 vs. 32).

There is one OIB Stage 1 (S1) multiplexer per wire driven from the tile, and we split these evenly between the two slices. Additionally, length-1 wiring is present on a single side. The total number of interconnect wire drivers *per slice* is 36, comprising:

- 4 of length $L = 1$ in the north, south and either east or west directions, $4 \times 3 = 12$;
- 4 of length $L = 2$ in all four cardinal directions, $4 \times 4 = 16$; and
- 4 of length $L = 6$ in all four cardinal directions, but shared evenly between both slices, $4 \times 4 \times \frac{1}{2} = 8$.

At Stage 1 (S1) of the *input*, we scale UltraScale's 64 logical multiplexers per slice to 32 and again keep the choice of using 7 : 2 shared multiplexer structures. This breaks an invariant in the UltraScale, where the number of IIB S1 multiplexers exceeds the number of IIB S2 multiplexers, so might require adjustment. Our design has disproportionately fewer wires than it has fewer CLB inputs and outputs, so the discrepancy must be adjusted for somewhere.

Table 4.9 compares the parameters of the UltraScale interconnect, as shown in Chapter 3, to the reduced version we have introduced here. In the following sections we will describe a circuit- and layout-generator for our architecture which can use the UltraScale-like topology generator.

4.10.1 Tile Generator

The `ReducedTile` class implements a BFG generator for a single tile of the Unnamed Columnar Architecture (UCA), emitting the circuit and layout for the design. Unlike most of the generators we have seen, this one explicitly expects it will be tiled next to multiple other instances of itself in order to form a complete fabric. Connections to adjacent tiles are by abutment, and connections to clock, power and ground are expected to be overlaid on top using metal layers above m_2. Unlike in the other BFG generators we have seen, the `ReducedTile` circuit is constructed in two parts: we start by assuming a 2-fly IIB and OIB structure but without knowing the exact connectivity of the constitutent multiplexers. That is imported from the external UltraScale-like topology generator. The types and quantities of multiplexers in each layer are parameters; once all of the physical multiplexeres are laid out, the imported netlist (circuit) is given to the automated router to lay all the intermediate wiring.

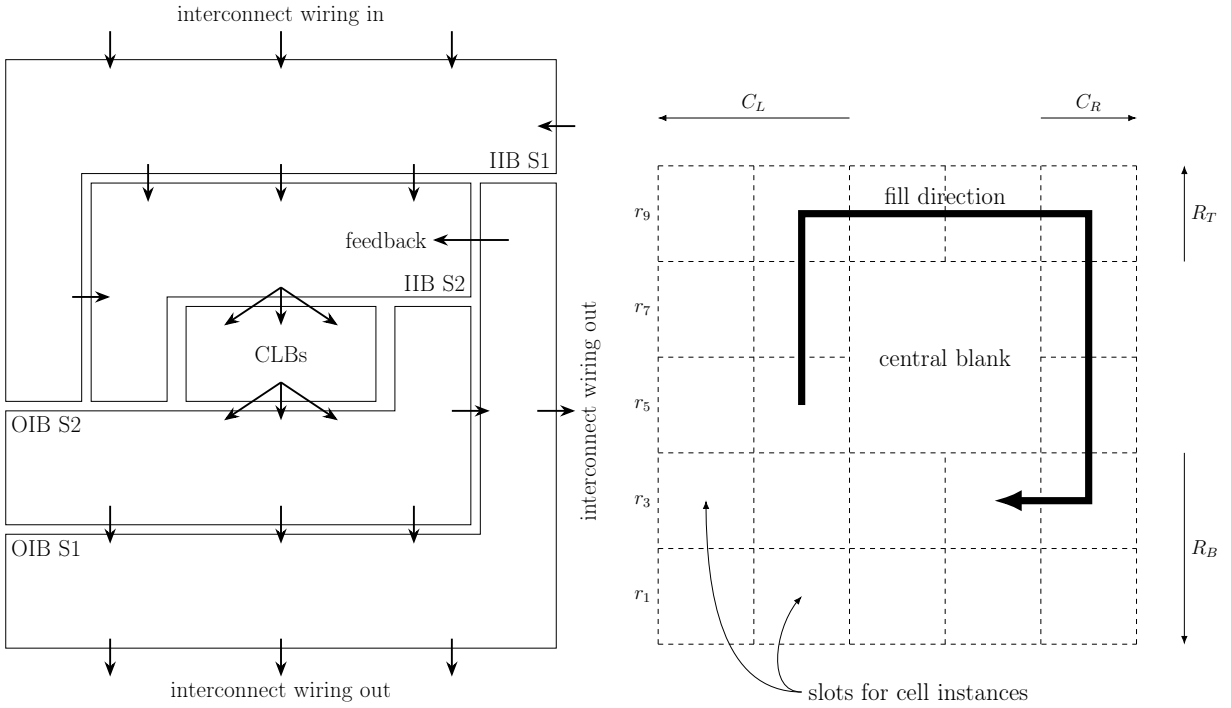
The generator follows a broad recipe for layout based around the key physical insights into the UltraScale architecture we saw in Section 3.2.1. We start with one of the two slices in each tile, as illustrated in Figure 4.52a. The CLBs sit at the core, surrounded by two layers of input (IIB) and two layers of output (OIB) multiplexers. The layers wrap around the core to moderate the worst-case path delay between an outer multiplexer and an inner one. Because we expect feedback paths from the OIB to the IIB, we also wrap the OIB layers back towards the input-side layers. Interconnect wiring is expected to come in from

Table 4.9: Comparison of UltraScale and UCA architecture parameters

Parameter	UltraScale	This Work
Stages	4	4
Wires per Bundle, N_W	8 (byte)	4 (nibble)
Bundles per Length, N_B	1	1
Wire lengths, L	1, 2, 4, 12	1, 2, 6
Terminating Wires per Tile of which north-south of which east-west	208 112 96	72 40 32
Max. Wires Through Tile (Vertical)	384	80
Wire pitch m_1, m_2 (nm)	-	340
Predicted Trunk Width (µm)	-	130.56
LUT Type	$k = 6$, fracturable to two $k = 5$	S-44, fracturable to two $k = 4$
LUT Bypasses	2	1
LUT Inputs (k + bypasses)	8	5
LUTs per Slice	8	4
LUT Inputs per Slice	64	40
LUT Outputs per Slice	32	16
IIB S1 Mux Type	2× logical 6 : 1 sharing one 7 : 2	2× logical 6 : 1 sharing one 7 : 2
IIB S1 Muxes per Slice of which Non-local Bounce	64 (logical) 4 (logical)	32 (logical) 2 (logical)
IIB S2 Mux Type	6 : 1	6 : 1
IIB S2 Muxes per Slice of which Bounce of which Bypass	64 12 4	48 6 2
OIB S2 Mux Type	2× logical 4 : 1 sharing one 5 : 2	2× logical 4 : 1 sharing one 5 : 2
OIB S2 Muxes per Slice of which Nonlocal Bounce	96 (logical) 5 (logical)	48 (logical) 2 (logical)
OIB S1 Mux Type	6 : 1 (8 : 1 for $L = 12$)	6 : 1
OIB S1 Muxes per Slice of which Bounce	104 15	36 7

× DRC errors ⚠ LVS untested

tiles/reduced_tile.h
tiles/reduced_tile.cc



(a) Broad layout recipe for the `ReduceTile` generator

(b) How `FillClockwise` allocates cells, with $C_L = R_T = 2$, $C_R = R_B = 1$

Figure 4.52: Preliminary machinery for laying out a tile

the north and to be driven out from the south and east. We "fill in" each layer with the appropriate number of multiplexers for each stage, numbering them as they lie, in keeping with our hypothesis that UltraScale multiplexer ordinals also correspond to physical position.

The layout generator is agnostic to the number of multiplexers in each layer, which are parameter inputs at instantiation. To make it easier to program the layout, we define a `FillClockwise` function that abstracts how successive elements are placed into each layer. The function successively allocates instances of cells to rows (`RowGuides`) around a central black space; see Figure 4.52b. The designer defines the height and width of the blank space; the number of rows above and below it R_T, R_B; the number of columns to the left and right C_L, C_R; and the types of cells to instantiate in each of the top, right, bottom and left passes. We invoke `FillClockwise` once per layer and align the resulting layouts.

The interconnect wiring trunk between tiles is handled by vertical and horizontal instances of `InterconnectWireBlock`. The vertical wiring is placed to the east and the horizontal wiring is placed to south of the slice layout in Figure 4.52a. The blocks are sized to match

the slice and the configured channel parameters. When the layout is tiled, the wire blocks will abut their counterparts and thereby connect corresponding wires. The calling generator, which is expected to repeat these tiles to form the fabric, is expected to configure wire blocks in each instance to break out the correct channel index. Recall that multiple long-spanning bundles of wires run in parallel: we must coordinate which bundle is connected to which tiles according to their position in the overall fabric.

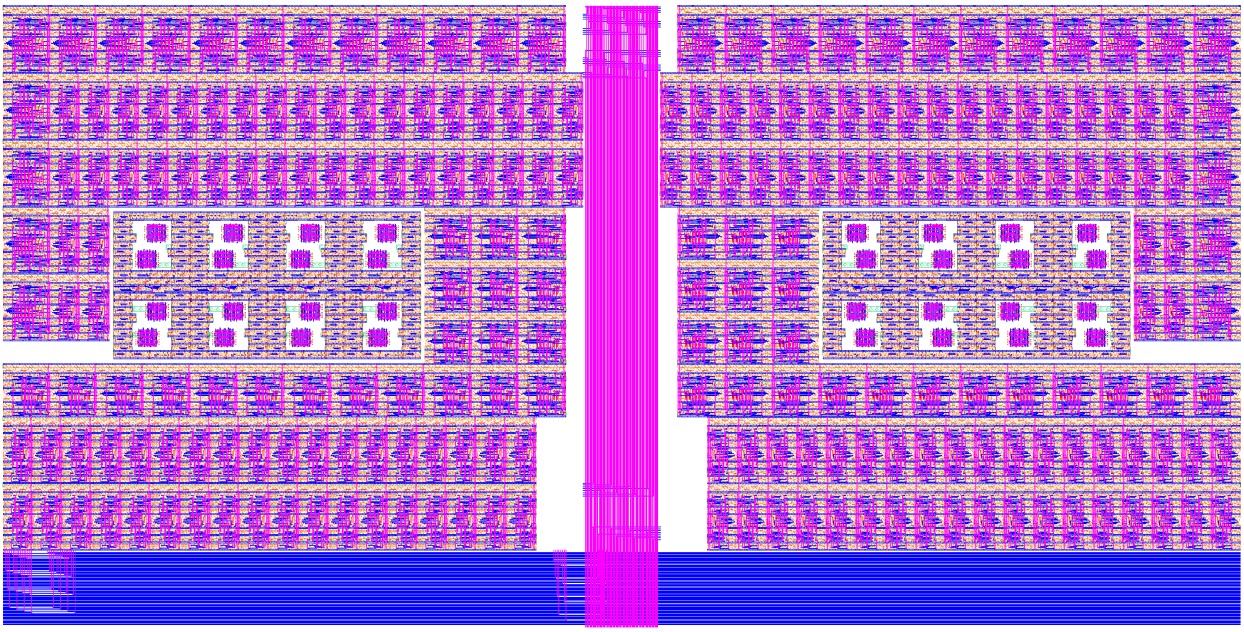


Figure 4.53: Layout of a UCA tile generated by `ReducedTile`, before automated routing

Once the slice is laid out and the trunk is configured, the slice is mirrored horizontally and added to the right of the vertical wire block. This is the tile's second of two slices. Mirroring ensures that both slices' outputs face the horizontal and vertical channels. The overall layout, before any of the internal connections are made, is shown in Figure 4.53.

To route the connections between the CLBs and multiplexers, and within the multiplexers, `ReducedTile` must first import the topology from the interconnect topology generator described in Chapter 3. At the time of writing, the topology generator emits a simple edge list containing the names of the source and destination instances for each connected output and input port. Port names themselves are generally not distinguished, since multiplexer output ports are distinct and input ports are interchangeable. Using BFG's `EdgeList` class, the `ReducedTile`:

1. configures the routing grid for the available routing area using the PDK defaults, submitting existing layout as obstructions;

2. reads the topology edge list;

3. maps instances referenced by name in the topology to instances within its own model, and thereby assembles an internally-valid list of required connections (Listing 4.9); and

4. and calls the router to solve for wiring (Listing 4.10).

Listing 4.9: Method to import generated topology, from `tiles/reduced_slice.cc`

```
std::vector<std::pair<std::set<geometry::Port*>, std::set<geometry::Port*>>>
ReducedTile::ExtractBFGInterconnectGraph() {
  std::vector<std::pair<std::set<geometry::Port*>, std::set<geometry::Port*>>>
      mapped_edges;

  EdgeList edge_list;
  edge_list.FromCSVOrDie(parameters_.edge_list_csv);
  for (const auto &entry : edge_list.edges()) {
    const std::string &source_name = entry.from().instance_name;
    const std::string &destination_name = entry.to().instance_name;

    std::set<geometry::Port*> source_ports;
    geometry::Instance *from = MapToPorts(source_name, false, &source_ports);

    std::set<geometry::Port*> destination_ports;
    geometry::Instance *to = MapToPorts(
        destination_name, true, &destination_ports);
    mapped_edges.emplace_back(source_ports, destination_ports);
  }

  return mapped_edges;
}
```

Listing 4.10: Invoking the router on the imported topology, from `tiles/reduced_slice.cc`

```
void ReducedTile::Route(Circuit *circuit, Layout *layout) {
  auto mapped_ports = ExtractBFGInterconnectGraph();

  RoutingGrid routing_grid(design_db_->physical_db());
  ConfigureRoutingGrid(&routing_grid, layout);
  RouteManager route_manager(layout, &routing_grid);

  for (auto &entry : mapped_ports) {
    if (entry.first.empty()  entry.second.empty()) {
      continue;
    }
    EquivalentNets net({
        (*entry.first.begin())->net(), (*entry.second.begin())->net()});
    net.set_primary(
        absl::StrCat(
            (*entry.first.begin())->net(), "_", (*entry.second.begin())->net()));
    route_manager.ConnectMultiplePorts(
        {entry.first, entry.second}, net).IgnoreError();
    LOG(INFO) << net;
  }

  route_manager.Solve().IgnoreError();
}
```

There are caveats. First, it is presently up to the designer to ensure that the naming scheme used for the topology generator matches that used for the `ReducedTile`. Second,

the scale of the problem is too great for the automated router under its non-linear cost routing model (Sec. 2.7.1). A linear cost model is partially implemented (or rather, it is fully implemented but does not yet work). Several optimisations are possible at this scale, and will be discussed in Chapter 5. Lastly, both the generated UltraScale-like topology and the cell layout bear co-optimisation, using for example the tuning techniques introduced in Section 4.4.2. BFG has afforded us the ability to validate (or invalidate) the educated guesswork of the Chapter 3.

Chapter 5

Discussion

We have presented BFG through its four principal contributions. First, in Chapter 2, we saw how BFG provides the machinery to create parametrised analogue/mixed-signal circuits by writing C++ programs, as well the mechanisms necessary to interoperate with third-party tools. Second, in Chapter 3, we demonstrated an algorithmic approach to building a modern interconnect topology by reverse-engineering the contemporary UltraScale architecture. Third, in Chapter 4, we first presented a catalogue of parametrised cells created in the BFG framework. We saw how the composition of these cells could yield increasingly complex — yet still flexible — instances of real FPGA circuits, such as the 4-LUT CLB, S-44 CLB and various configurations of one-hot transmission-gate multiplexers for interconnect. Fourth, in Section 4.10, we proposed a modern, columnar reconfigurable fabric architecture (UCA) based on our insights into the UltraScale and key results from literature. We demonstrated how the high-level topology generator could be combined with our catalogue of cell generators to produce a complete tile for the UCA fabric.

FPGA generation was conceived as a labour-saving methodology for design-space exploration (Sec. 1.7.3), for research and innovation, and eventually for rapid fabrication. Popular "top-down" standard-cell synthesis of FPGA fabrics has readily met these goals, but at the expense of circuit performance. We claimed that a better way to build FPGA generators was to build circuit and layout directly, using a hybrid, programmatic custom design flow such as BFG's. Supporting this, our results found that BFG's 4-LUT CLB achieves 59% better power-delay-area product over a logically-equivalent circuit synthesised from standard-cells using state-of-the-art tools. Moreover, we showed by sweeping transistor sizes in the read path of the 4-LUT CLB that 1) BFG makes circuit tuning a simple process that can easily be automated; and 2) circuit tuning can have a significant impact on performance. We expect that the design — which is currently a collection of lightly-modified standard cells and two passive transmission-gate multiplexers — could be improved much further. At near-100% placement density and using immutable cells, we do not expect that the standard-cell design could be improved in kind.

We do *not* claim that BFG's generators can outdo an expert human (or team of humans) hand-tuning a circuit, as would be economically justified for a flagship commercial FPGA

product. Still, any circuit advantage BFG achieves it does in tandem with being a generalisable automated tool; like existing FPGA generators, it supports labour-saving design-space exploration, and research and innovation. It will eventually support rapid fabrication. BFG serves as a repository for good (and bad) circuit and layout ideas; the BFG AMS framework enables us to program designs ranging from extremely general to extremely niche. Once a generator is programmed to generate high-performance layout, the insight and know-how of the designer are codified for all to use. We have shown many examples of how a single well-designed cell can be re-used to improve others through composition, with exceptional structures being added as necessary (Secs. 4.7, 4.4). We even showed (Sec. 4.3.9) how designs can be trivially ported to different processes within the same process class. When a user fine-tunes a generator, they do so for many more use-cases than just their own. The power of software-defined layout is in its ability to capture designer intent, and to then foster re-use, rapid iteration, and refinement.

With that said, this work has also highlighted the inescapable challenges of programming circuit and layout generators. Chief among them is the need to manage the complexity of each degree of freedom added by a parameter. We can now appreciate that building a robust layout generator is best served by upfront consideration of *all* future parameters. In the `Sky130Mux` (Sec. 4.3.8), we saw how the completely free-form placement of six measly transistors quickly compounded into overwhelming complexity. The first reason for this was that the designer (your author) did not understand what they were generalising before they tried to generalise it[1]. The second was that C++, venerable as it is, makes expression of the constraints at-times unreadable — having to manage a `static_cast<int64_t>` in the middle of a long arithmetic expression is unhelpful. When the second and third metal layers were added, all of the past assumptions about layout had to be revisited, remembered, and re-integrated into a more complete view of the overall constraints.

The best solution to overwhelming complexity is, as always, to decompose the design *ad absurdum*. We saw this work for the $N:1$ multiplexer in Section 4.7: at the core, a basic `Sky130SimpleTransistor` offers a highly-parametrised generator that supports everything we have ever needed from a transistor layout. Composing these into transmission gates and then composing the result into a transmission gate stack are two relatively simple exercises that layer complexity manageably atop one another. The very act of *defining sub-generators* forces the designer to consider the interfaces between constituent elements: one must establish what externalities can impact an inner design and what must be known in advance to account for them. The number of surprises, and revisions, are thus reduced.

Relatedly, there remains the question of which software patterns are the most effective for writing circuit and layout generators like BFG's (Sec. 4.1). We expect that with time and development effort more and more useful abstractions will be discovered, and that the *most* useful will jostle for rank. All of the existing utilities in BFG — the transistor abstraction, the `RowGuide` row, `MemoryBank` of rows, saving named points, and so on — were born of necessity in an effort to reduce toil. Some ideas, like creating a right-angled wire, gain

[1] "Premature generalisation" is an established software anti-pattern.

moderate adoption; other ideas, like a simple function for filling empty row space with decap cells, become nigh ubiquitous. Eventually we will have evidence for which are the best. Not just from BFG, but across the entire community of circuit designers trying to express their layouts with whatever frameworks they choose.

Enabling a community of designers to contribute to every circuit generator is a foundational aim of this work. As introduced in Section 1.5, the value of open-source software is now well established within the realm of computing. By giving hardware designers the same ability to codify, share, iterate and refine layout as code, we repurpose this now-common facet of software-development for the robust definition, and refinement, of high-performance circuitry. In keeping with its aim BFG is free and open-source software (FOSS), unencumbered by any restrictive licenses. BFG's *dependencies* are all free and open-source, too.

A valid criticism of this work is that all of the generators we have demonstrated were built for the SkyWater 130 nm process, a legacy technology that — despite continued relevance for some contemporary products — is about 25 years behind the state of the art. That is true. Recipes for layout generation targeting newer processes will require additional sets of constraints, or completely new approaches (Sec. 2.2.1). We believe, however, that this misses the point. SkyWater 130 nm was the very first *production* process to be made open-source. Users can freely inspect its rules and contribute to our designs without burdensome three-way NDAs. Users are not siloed, not forced to reinvent the wheel for lack of access to any existing body of work. This is a boon for the democratisation of circuit and architecture research. While there *are* other educational and research-oriented PDKs [96], [106], they are not backed by a foundry capable of manufacturing one's final designs. SkyWater 130 nm is: any of BFG's designs can be taped-out (though not all of them will work). BFG is available immediately in its entirety for any researcher, enthusiast or practitioner to download and contribute to.

Another valid criticism of BFG centres on design verification, where standard-cell FPGA generators have a significant advantage. The latter produce regular digital circuits from pre-characterised cells which the user can then validate through a largely automated (but still non-trivial) set of tools. BFG, on the other hand, can rapidly generate many different circuits and layouts that each need to be validated from the ground up. To a digital designer BFG adds to the verification burden rather than eases it. On one hand this is also true: BFG's cells must be validated as any custom AMS circuit would; on the other, because all of BFG outputs are defined in software, it is possible to automatically generate the SPICE simulations necessary for characterisation in tandem with each design. The tuning and performance experiments (Sec. 4.4.3) demonstrate a manual approach to this; in later work the pipeline will be automated. Better yet, once specific design points are validated they can be stored as known-good examples, used by others verbatim (as is common with SRAM generators), and even tested by Continuous Integration (CI) systems for regression.

The prerequisite work to allow BFG to interact with third-party tools was a significant challenge (Sec. 2.8). The software ecosystem — if it can be called that — for the chip design industry, across both AMS (Sec. 1.3.3) and digital (Sec. 1.3.2) ICs, is staggeringly difficult to interact with. Though the tools should be lauded for what they accomplish,

three decades of near duopoly have resulted in de facto standards for information exchange built around proprietary formats (LEF/DEF, GDS), fragmented and incompatible dialects (flavours of SPICE), and a culture of gatekeeping the arcane knowledge required to unlock basic features. A new so-called "OpenAccess" format requires *annual membership dues* for access[2]. The help manuals are thousands of PDF pages and are inaccessible by any modern search engine. Version control is an afterthought, if it even exists. There are few accessible software interfaces, let alone standard APIs. Secrecy is a large, and understandable, factor; as is aversion to technical risk in such a capital-intensive endeavour. Another factor is simply that heroic engineering (and market demand) will see that the chips get built in spite of these challenges. Yet, compared to the software industry that has matured around it, the design process for chip designers feels patently archaic. We touched on this when we introduced BFG's microservices for VLSI (Sec. 2.9), and we will discuss it further in the next section. BFG should not have needed to build an AMS framework from scratch: the literature existed, the know-how existed, and yet none of it was available. What a *relief* it is, then, that BFG can now interact with third-party tools but that, thanks the VLSIR framework, it need not be a concern for BFG's circuit designers.

In conclusion, BFG has shown that building a bottom-up silicon-compiler for FPGAs can yield significant circuit performance gains across power, area and delay over top-down standard-cell-based approaches. It has also shown the (perhaps predictable) complexity in getting such a system to work. To be a viable alternative, BFG — or another tool like it — will need the contributions of many researchers, enthusiasts, and interested practitioners to build atop one another. This was the model that worked for GNU in the 1990s and has worked for large open-source software projects since, and it can be brought to bear again here for the good of researchers, engineers, computer architects, digital-circuit designers and analogue-circuit designers. BFG has provided the framework and a series of examples so that no one else has to start from scratch.

5.1 Future Work

5.1.1 Quality of Generated Designs

BFG generators for FPGA cells, described in Chapter 4, are ripe for improvement. As we have discussed, a central aim — and hope — for this work is to enable others in the community to contribute these improvements directly to the framework. A pressing upgrade, for example, is to create versions of our cells that use latches and/or SRAMs for configuration memory instead of flip-flops. An SRAM-based CLB will require more configuration wiring but will be substantially smaller than the existing flip-flop based approach. A smaller design should reduce overall wire-length, and thereby wire delay. Fewer transistors should result in reduced static, if not overall, power consumption. The CLB also has a lot of empty

[2]Academic membership is free, but is restricted to universities and comes with the caveat that "Academic members [...] are not granted rights to vote on any item under consideration by Si2." [91]. This is not "open".

space around its central multiplexers, suggesting that there is room to shrink the overall cell. We should evaluate, for instance, if it is worthwhile to replace the `Sky130Mux` with the `Sky130TransmissionGateStack` and a decoder, or another multiplexer structure altogether.

5.1.2 Design Verification

Characterisation

BFG's internal model already contains all of the information necessary to set up complete SPICE simulations of a design at run-time. These simulations should be used to automatically characterise every BFG cell, from scratch, based on its particular parameter values. This will involve, for example, measuring the relative delay between input and output pins under different assumptions about the input signal shape and the output load; measuring power use, including under different assumptions about transistor activity factors; circuit performance (and correctness) at each of the process corners; and monte-carlo simulations, where they are made available by the PDK, to understand circuit behaviour with manufacturing variance.

The problem of automatically generating logic tests for unknown (*blackbloxk*) circuits is non-trivial, and yet it would be very convenient for BFG to be able to automatically synthesise appropriate test-benches for any design. If the designer can be called on to specify the relationships of the pins as part of the cell's construction, it should be possible to deduce how pins relate transitively through multiple child cells and to therefore craft appropriate stimuli for each output. More generally, Design For Test (DFT) methodology can be applied to the creation of BFG generators themselves, supported through extensions of the BFG libraries. Higher-level generation frameworks, like Chisel for RTL, provide examples of how this can be built-in. Just as circuit and layout design patterns will accumulate, so too will patterns for testing and characterisation; future designers will have a suite of templates to choose from to make their lives easier.

Open-source SPICE simulators like Xyce [115], which are not restrictively licensed, can then be used for rapid and continuous characterisation as part of the BFG design cycle. BFG could, for example, invoke Xyce to run the generated test-benches at run-time and then collate the results into a complete report. A new VLSIR schema for characterisation data could then be created, together with an appropriate conversion tool, to automatically produce a standard Liberty-format description of the cell.

Layout vs. Schematic and the Tyranny of Someone Else's Tools

LVS should remain an external function since it serves as a third-party check of BFG's work. Currently, however, the flow is cumbersome. An unfortunate feature of the VLSI "ecosystem" is that the same tools, operating on the same technology node, can expect widely different inputs depending on who has set them up. Among the configurations of various SkyWater 130 nm tools, for example, some are set up to expect primitive transistors that match

the foundry models, while other are set up to expect their own abstractions. Our Siemens Calibre[3] setup required us to replace the SkyWater 130 nm models with bespoke definitions: `nshort` for short-channel NMOS devices instead of `sky130_fd_pr__nfet_01v8`; `pshort` for short-channel PMOS devices instead of `sky130_fd_pr__pfet_01v8`; `phighvt` for high-threshold-voltage PMOS devices instead of `sky130_fd_pr__pfet_01v8_hvt`; and so on. Elsewhere, some tools would expect MOSFET parameters to be scaled by 1×10^6, while others would not. Some simulators (HSPICE, Spectre) provide mechanisms to scale these for a whole file while others (Xyce) do not. BFG and the various VLSIR tools should make designers' lives easier by being configurable to match these output quirks on a per-tool basis. In lieu of that, the current scripts do most of the work.

Integrated Design-Rule Checking

As we have seen, DRC checks are a core part of the design loop when one is trying to program layout that is correct by construction. BFG operates on a simplified set of DRC rules (Sec. 2.2.1) and relies on the designer to check for correctness with an external tool. BFG should include a minimal DRC checking mechanism that aids the designer by flagging violations during compilation. Any such engine would also serve as the foundation for more automated layout features. Additionally, we expect that the types of DRC rules BFG supports will expand as necessary to support more diverse designs across more and more processes. It would be very convenient for BFG, either directly or through VLSIR, to support reading the DRC rule decks of common FOSS tools like Magic [72] and KLayout [49]. It would also be helpful to support the rule formats of common commercial tools, but they are both proprietary and many in number. It sure would be nice if someone created a single, structured, language-agnostic schema for them with VLSIR instead.

5.1.3 Automated Routing

Shortest-Path Algorithms

One of the most complex components of BFG is its automated router, and it stands to be improved in several ways. The first of these, and the easiest, is to use slightly more advanced shortest-path algorithms. For example, it would be trivial to switch from Dijkstra's shortest-path, the implemented algorithm, to the A* algorithm. A* generalises the cost of a path to

$$f(n) = c(n) + h(n)$$

where n is the next node, $c(n)$ is its classical cost in Dijkstra's original, and $h(n)$ is a heuristic we introduce to more selectively select the next candidate. The existing priority queue in our implementation can simply be updated with the new cost function $f(n)$. Because we are

[3]Siemens Calibre can be used for DRC checks, LVS checks and parasitic extraction.

operating on a regular grid, we can choose $h(n)$ to be the Euclidean distance from node n to the search target. At the same time, other algorithms should also be evaluated: double-ended Dijkstra; Bellman-Ford (which handles negative weights); and so on.

Relatedly, our approach to finding paths for multiple terminal points needs revision. Recall that we solve this by first finding the shortest path between two points, and then finding the shortest path from each successive terminal to any point along the existing path. The real solution to this problem is to find a minimum rectilinear Steiner tree (MRST) [89], which is an NP-hard problem. Heuristic approximations exist and should be experimented with. Single-trunk Steiner trees, for example, are not dissimilar to our hack and may be found in $\mathcal{O}(n \log n)$ time.

Optimisation

A fundamental weakness of the router is that it finds shortest routes in the order that they are specified and does no meta-optimisation of the solution set. Congestion is handled poorly as a result. A straightforward modification would be to add weights to edges and vertices in congested paths, then re-run routing around the new costs whenever paths are found to collide. This should distribute paths around congested areas; it is a form of the Rip-up and Reroute technique that more robust algorithms employ. The problem of VLSI routing is well-studied. Though our goal was to make a *simple* and *fast* inline router, it is now clear that even BFG's designers could benefit from robust larger-scale routing techniques when their designs are complex enough. It would be worthwhile to integrate and support, as an option for designers, external FOSS routers such as TritonRoute [47].

Smaller scale routing problems, on the other hand, could be handled by numerical optimisation techniques such as Integer-Linear Programming (ILP), Mixed-Integer Programming (MIP), Quadratic Programming (QP) and even Boolean Satisfiability (SAT). In their work on Bonncell [43], [104] the authors show that these techniques (MIP in particular) can feasibly yield an optimal set of routes for small problems. They showed that for problems of ~ 10 transistors the router completes within a minute and could subsequently be used inline, and for problems of ~ 100 transistors the technique works but could take over an hour. An implementation for BFG should afford some relaxation of the problem constraints, since the designer controls correctness up to the routing step; these techniques might offer the designer a quicker solution, depending on their scale, and should be integrated into BFG. In fact, the entire BonnCell approach to cell generation (which includes automatic transistor placement) could be integrated to generate subcomponents for larger BFG designs.

Revision of Dubious Features

The alternative routing approaches discussed so far have in common that they only support linear-cost models. Recall that the BFG router builds (Sec. 2.7.1) an $N \times N$ grid of vertices with $\mathcal{O}(N^3)$ edges in order to support *non-linear* wire costs. Linear models have $\mathcal{O}(N^2)$ edges. We have no evidence that the additional complexity has been worth it. On the

contrary, we have a lot of evidence that the router is too slow and does not scale well: the `ReducedSlice` (Sec. 4.10.1) cannot be routed yet. BFG should complete its linear-cost model and a study should be performed to evaluate what, if any, performance benefit the non-linear model can provide.

The ability to access off-grid ports is another feature of dubious utility. Though well-intentioned, the result of allowing connections to off-grid points is usually the painting of oneself into a corner. Combined with the lack of a higher-level optimisation loop and the significance of route ordering, the off-grid edges and vertices often create more obstacles to routing than they solve: their neighbours can seldom be routed and wiring area is wasted. This is solved by finding sets of mutually correct routes, with simultaneous objectives: in order to fully realise this feature in the interest of the designer's convenience, the router must include some of the aforementioned optimisation techniques. What's more, the two methods that BFG uses to connect off-grid points (Sec. 2.7.1) are heuristic and error-prone. Instead, the major routing grid should be augmented with a very-fine-grained routing grid near the off-grid ports so that they can be connected using the existing shortest-path algorithm, and not in three separate steps.

Performance

The router supports multithreaded path-finding, but synchronises on any update to the state of the central routing grid. This typically happens when paths are installed. It also happens when off-grid points create new edges and vertices, though it need not. The off-grid points and their related objects should be treated as ephemeral by each path thread and made permanent only when the path is accepted. There are also scenarious when mutliple paths are highly-unlikely to conflict. In the `ReduceSlice`, many routes between IIB and OIB layers will not come near each other. It should be possible to use a bounding-box around the start and finish points to determine which routes are unlikely to conflict, to let them run in parallel, and only check for collisions for final correctness. Deferring correctness in this way adds too much overhead if there are frequent path collisions.

Lastly, two additional features would make the router easier to use on a daily basis. First, determinism is an extremely useful property for debugging and for designer sanity. If a correct solution set is found it should be reproducible. Any intentional randomness should be controlled by a seed value that is reported to the user, and that can be overridden by user input. Unintentional randomness, such as in data structures sorted by pointer value (that is, memory addresses), should be replaced. Relatedly, the second feature is to provide the designer with more complete debugging information. Often a route will fail and it will not be obvious why: visual cues, such as the existing ability to export the state of routing-grid vertices to the layout, help a lot. Additional and in-depth traces of router reasoning will help designers avoid unsatisfiable requests. It is useful to inspect, visually, what paths or obstacles invalidated routing objects and at what point in the process.

5.1.4 Clock Tree Synthesis and Power Strap Generation

In a typical digital VLSI flow (Sec. 1.3.2), Clock Tree Synthesis (CTS) is performed after cells are placed and general (global and detailed) routing is complete. The goal of this step is to create a fractal tree structure that connects a clock source to all of the circuit's clock inputs, in such a way as to minimise — or, at least, to control — the skew between the signals arriving at each input. BFG should implement CTS. For a regularly tiled structure such as an FPGA, it is common and straightforward to build an H tree. For more general circuits, BFG can defer to an existing FOSS solution such as TritonCTS 2.0 [39].

Power strap generation typically follows CTS. Power straps are more broadly known as the Power Distribution Network (PDN): wiring which connects the power rails of each cell to upper-layer power distribution layers, taking care to minimise resistive losses where possible, usually by using the upper layers in the technology stack where metal wires can be much wider (in SkyWater 130 nm these are metals m_4 and m_5). PDNGEN is an existing FOSS tool that implements such a generator, part of the OpenROAD set of tools [1]. PDNs are regular grids of alternating power and ground metal wires that are parametrised by the designer; BFG should implement PDN generation, either by calling out to an external tool like PDNGEN, or on its own.

Note that, even without built-in CTS and power strap generation, designers can always import BFG designs into an existing digital IC tool flow to perform these steps. Since BFG's designs can be converted into industry-standard LEF/DEF and GDSII formats through VLSIR, they are no different to any external macro that one imports into a typical digital floorplan. Nonetheless, adding support for these tasks directly into BFG will greatly simplify the design cycle and thereby ease the designer's burden.

5.1.5 Geometric Engine

BFG implements a minimal set of geometric routines to support rectilinear IC layout. As discussed in Section 2.3, there is no real value in re-implementing these routines beyond 1) trying to determine what this minimal set is, and 2) providing a simple, self-contained, and freely-licensed library. In reality, the large body of work on computational geometry should be readily drawn on to provide more robust and complete routines to the BFG designer. If not, the geometric routines within BFG should at least be further developed to support common operations on a broader set of shapes. As a starting point, Table 2.2 should be completed. BFG will have to adopt the invariants and assumptions made by de-facto standard implementations, such as those in CGAL [99]. Once it does, it it will be able to re-use common algorithms from literature, such triangulation for simple polygons.

The principle benefits of robust geometric routines are in the affordance of convenience and precision to the designer. BFG cannot currently intersect two polygons, for example, but being able to do so will allow for more compact layout of interesting structures. It should also be possible to quickly merge, intersect and mutate any two shapes without having to worry about which operations are supported by which types, and what relaxations

are possible without sacrificing correctness. Elsewhere, spatial indexing should be used throughout layout management routines to optimise collision detection. Currently, `Layout` stores all shapes by their layer only. There is a rich body of work on spatial data structures to consider [86]; the router already uses a k-d tree to manage off-grid points (Sec. 2.7.1). Limiting the search space for collision detection, for example, might be better served by a quadtree.

Note that, in general, BFG must negotiate many shapes and their interactions, both according to a central set of rules and specific sets of rules among each pair of shapes. Video game engines are scalable and fast software frameworks that do just that. They, too, must constantly manage the relationship between many *entities* and their environment according to sets of rules, or *systems*. The Entity Component System (ECS) architectural pattern was developed to address these requirements. BFG does not have the same real-time processing requirements as a video game engine, but it does require greater precision; even if it may not be possible to re-use a game engine directly, the suitability of the ECS pattern to IC layout should be investigated. ECS engines have already been successfully implemented for other 2-D [12] and mechanical Computer-Aided Design (CAD) software [5].

5.1.6 Usability and Scaling

Most of the other improvements to be made to BFG concern its *ergonomics* from a circuit designer's point of view. As we have stated, we expect the most useful and convenient of these ideas to become apparent through experimentation and use. It is already apparent, for example, that the designer has to worry too much about managing the lifetimes of cells in their design. It is easy to inadvertently recreate the same cell multiple times. The library should cache generated cells by a hash of their parameter structure, so that requests to generate an identical cell can be served from the cache instead of being generated from scratch. Since this would apply to any child cells in a design, it would greatly speed up the composition of large sets of cells which vary only slightly from instance to instance. The `ReducedTile` is a good example of this (Sec. 4.10.1): aside from the differing `InterconnectWireBlock` parameters, most tiles in a large fabric — and therefore most *components* of those tiles — would be identical.

Though we offer many examples of how BFG can be used through the cells in Chapter 4, it would be useful to document and maintain a set of best practices as they are discovered. Some design patterns are not *obviously* good or bad until after some trial and error. As an example, consider that the first BFG generators were designed to avoid keeping any of their state during construction. This has the nice property that, once configured, a single generator can be used multiple times by many independent threads with no risk of contention. It would not be possible to leave a generator in a *bad* state, so it can be long-lived. The disadvantage is that the code is vastly more complicated, since now every method requires an additional argument to access state from some earlier part of the computation. In reality, "generators" should be single-use objects that represent the generation of a single cell; we never need to generate the same cell twice. Even then, a *factory* pattern could be used to present the

creation of these generators statically, if needed. As an added advantage, a stateful generator can be queried for calculated properties of its design after it is finished. More powerfully, it could be made possible to *revise* a design after the fact and have it regenerated in-place later. This could be very useful for negotiating constraints with other cells.

BFG's layout guides (Sec 2.3.4) and the `PolyLine` (Sec 2.3.3) are great examples of how an apt abstraction can greatly simplify a layout problem. Many more abstractions exist that might benefit circuit and layout designers, and we need only look to other software domains for examples. Software engineers for mobile and web-based applications, for instance, are provided many conveniences for arranging elements within a user interface. The approach taken in these fields shows us that general constraint solvers make programmatic layout generation markedly simpler. A mobile developer might only care that one element is "left of" or "below" another, or that that it is "equally spaced among" several others, and their library will support them expressing this (for Apple iOS devices, this could be with SwiftUI [44]). The designer might also care that two still-different elements are separated by an exact distance, and their library will support that too. BFG should allow the designer to express varying degrees of constraints in the same manner, and should implement the same general constraint-solving methods that they do. Web browsers, for example, include complex (and fast) constraint-solving solutions for placing under-, over- and exactly-specified shapes. The authors of the Cassowary [6] linear arithmetic constraint solving algorithm even used layout (of a user interface) as a motivating example.

Domain-Specific Language

The language for expressing geometric constraints is at the heart of generating layout. It need not be the same language as is used to implement the underlying engine. C++ is a verbose language whose logistics (strong types, casts and memory management) can easily obfuscate the underlying geometric relationships being expressed. Consider the method in `Sky130SimpleTransistor` that computes the required extension of the polysilicon layer over the diffusion in the transistor it generates. The code is relatively short (Listing 5.1), but as many (or more) lines are spent explaining the geometric reasoning behind the code in comments (Listing 5.2). The core expression is shown on lines 475 – 478 of Listing 5.2. Similar expressions are used in comments throughout BFG code for a variety of generators, expressing the minimums and maximums of various quantities, as well as other invariants. The quantities are often nested. In general, the mathematical expressions provided for layout constraints make it much simpler to follow the reasoning behind various calculations. This, in turn, makes it much easier to determine what needs to be modified when revisions are needed. It has been *much* easier to maintain code that details its geometric intuition than code that does not.

Listing 5.1: Computing extension of polysilicon over diffusion in `atoms/sky130_simple_transistor.cc`

```
507 int64_t Sky130SimpleTransistor::FigurePolyDiffExtension(
508     int64_t separation_to_metal_via_centre,
```

```
    bool allow_horizontal_metal_channel) const {
  const PhysicalPropertiesDatabase &db = design_db_->physical_db();
  const auto &metal_rules = db.Rules(FirstMetalLayer());
  const auto &pcon_rules = db.Rules(PolyConnectionLayer());
  const auto &diff_dcon_rules = db.Rules(DiffConnectionLayer(), DiffLayer());
  const auto &metal_dcon_rules = db.Rules(
      DiffConnectionLayer(), FirstMetalLayer());

  int64_t required_metal_spacing = allow_horizontal_metal_channel ?
      RequiredMetalSpacingForChannel() : metal_rules.min_separation;

  return required_metal_spacing - (
      diff_dcon_rules.min_enclosure - metal_dcon_rules.via_overhang +
      separation_to_metal_via_centre - pcon_rules.via_height / 2);
}
```

Listing 5.2: Comments for computing the extension of polysilicon over diffusion in `atoms/sky130_simple_transistor.cc`

```
//     |       | licon   |           |           |
//     |       | via     |           |           |
//     | diff  +---------+  ^        |    poly   |
//     |     ^ | li1 pour|  | A      |           |
//     |     | |         |  v        |           |
// y=0 +-----|-|---------|-----------|-----------|----------
//         B v +---------+           |           |  ^
//                                E  |           |  | we compute this, E
//                                   |           |  v
//                              +-----+           +-----+
//                            ^ |     +-----------+---------
//                        C | | |     |  li1 via  |     |
//                          | |   D |           |     |
//                          v |     |           |     |
//
// The calculation is symmetrical in y, so it's ok that we only consider the
// case where the tab is below the diffusion.
//
// A: min. enclosure of diff on licon (diffcon) via
// B: li overhang of licon via
// C: length of poly beyond extension to midpoint of next li pour
// D: height of next li pour above its midpoint
// E: spacing between diff and poly edge we need to figure
//
// Q: required min. separation of first metal layer (li, local interconnect,
//    typically)
//
// (A - B) - (-E - C + D) = Q
// E + A - B + C - D - Q = 0
// E = Q - A + B - C + D
//   = Q - (A - B + C - D)
```

Furthermore, recall that one of BFG's core aims is to enable designers to *communicate* their designs. In order to reduce the maintenance burden and to simplify the sharing of layout ideas, BFG should support more "readable" languages for expressing layout — or for writing generators altogether. Python is an obvious first choice: it is widely adopted and understood, avoids a lot of logistical verbosity by being dynamically typed, and prizes expressivity through its idioms [77]. GDSFactory [100], an AMS framwork in Python, provides

a good example of how comparatively succinct layout recipes can look. As a first step, BFG could provide a Python interface to its existing C++ library through the Simplified Wrapper and Interface Generator (SWIG) tool. C++-defined generators and Python generators could thereby co-exist.

Next, a Domain-Specific Language (DSL) is needed to express geometric reasoning in a way that can be consumed by both humans and automated constraint solvers. The simple mathematical expressions in commented throughout the BFG source code are an instructional basis for the kinds of expressions that would commonly be needed. The DSL should: enable one to express the constraining relationships between quantities using shorthand (or references); use references for PDK rules, and perhaps have some inferred automatically; support common mathematical operations, as well as max, min, arg max, arg min, floor, ceiling, et cetera; and support common geometric operations such as midpoint, bisection, et cetera. There are many existing DSLs for expressing exactly these kinds of relationships, but they differ in their intended audience. Many are intended for automated problem solvers, in particular for LLM-based methods. The work by Cascaval et. al. [14] is particularly relevant to CAD applications such as BFG and would serve as a good starting point. The DSL could even extend the Python language to embed directly into a more general program for crafting the generator, as in [70].

Graphical User Interface

To visualise the output of a BFG generators, designers are currently required to convert their outputs to a compatible format and use third-party layout viewers, like KLayout [49]. This process has three disadvantages:

1. The intermediate format, typically GDSII, does not accommodate design metadata. GDSII was created by the Calma Company in 1978. There is no easy way to associate the generating source code for an object with how the object is presented. It might be possible to address this with a kludge like packing debugging symbols into text annotations, but we are fundamentally limited by the design of the intermediate format.

2. The process is cumbersome and manual. The designer must compile and run BFG, then run the conversion tool, then update their viewer to display a new version of the layout. Here the kludge is to use a filesystem watcher with a script to automatically trigger compilation, generation and the viewer update, but in our experience this is not a robust solution.

3. The barrier to first visualising the layout is high. A technology model is required and must be shared with the conversion tool. The layout viewer must be also be configured. In order to generate an output, the BFG generator must implement a minimal interface. It is not possible to visualise half-baked layout ideas on their own.

BFG should provide a built-in Graphical User Interface (GUI) for viewing and inspecting layouts. The GUI, which should still be optional, will not replace the use of external viewers,

but will provide a convenient way to visualise layout recipes as they are being developed. A designer should be able to view their (half-baked) BFG program and its layout side-by-side. It should be possible to quickly identify which parts of the code led to the generation of which drawn shapes, and to jump back and forth between source and layout. The designer should also be able to visualise PDK design rules and operative constraints, such as the minimum separation from other shapes, and which shapes are closest to violation. Visualisation of the router could include measures of congestion (heat maps), paths that have been tried and failed, and other hints for success. These features common to many existing systems where layouts are programmed, most commonly in Integrated Development Environments (IDEs) for mobile applications and dissertations written in LaTeX.

5.1.7 VLSI Microservices

In Section 2.9, we discussed how "microservices" could benefit the IC design flow by asynchronously offloading computationally intensive tasks to dedicated machines. We showed how BFG's router and, separately, SPICE simulators could be hosted behind RPC interfaces that client code can invoke at will. These interfaces, which provide structured schemas for invoking each service, can become the standards for multiple underlying implementations. In addition to better resource allocation, distribution of IC design tasks simplifies client code and enables greater automation.

Accordingly, microservices should be developed for LVS, DRC, characterisation steps and any other commonly-used external tools. BFG should then use these services to provide results to the designer. We imagine, for example, that the designer can be given simulation feedback in a GUI as soon as it is available. Given an appropriate branching mechanism, multiple ideas for a design could be simultaneously simulated in the background (on remote machines) and presented to the designer for comparison. Characterisation could be included in design generation without additional set-up or tool installation by the designer.

5.1.8 Tape-Out and Architecture Evaluation

BFG needs to be proven in Silicon. Once the router has been upgraded to scale and the `ReducedTile` completed, a trial fabric should be manufactured and tested. As discussed, CTS and power strap generation are not prerequisites since we can import BFG's fabric as a macro to an existing tool flow for those steps. With a complete fabric, the Unnamed Columnar Architecture should also be evaluated *as an architecture* using the standard (MCNC '91 [116], VTR [26] and OpenCores [71]) benchmarks for FPGA performance. With a completed chip we will be able to, and therefore should, compare the realised performance of the device against our simulation results and against other *real* FPGAs.

5.1.9 AI/ML Integration

Machine Learning (ML) methods from the field of Artificial Intelligence (AI) hold significant promise for the automation of circuit design and IC layout. In Section 1.7.4 we summarised how traditional numerical optimisation techniques, ML/RL and, most-recently, LLMs are actively being investigated in the hope of fully automating the design of analogue and digital ICs. It is reasonable to ask what place, if any, semi-automated AMS frameworks like BFG have in such a future.

The first role that BFG and other AMS frameworks play is as engines for composing designs into complete, manufacturable blueprints. The central limitation of the automation techniques we have seen so far has been scale: the target designs (individual standard cells, op-amps, and so on) usually have less than 100 transistors. Until a scalable system for end-to-end automation can be developed, one of the most useful tools we have for building ICs will remain the hierarchical composition of larger designs from smaller ones. Automatic techniques can already be used to implement the basic cells (Sec. 4.3) from which BFG's larger cells are made. Recall also that a key motivation for this work (Sec. 1.5) was to enable automatic FPGA generation while undoing the abstractions that hindered performance (standard-cells). We should not rush to wholesale replace every class of circuit with a newly-automated version *until* it is shown to be superior. Within BFG, any number of generator versions and ideas can co-exist; the best implementation can be enabled whenever it applies.

BFG's second role can be as an engine for experiments, first by humans but eventually by other machine systems and agents. Combining LVS, DRC and simulation into a single programmable tool should provide a modular, extensible and fast framework for machine-driven experiments on layout and circuit design. Consider the iterative step in an RL loop, which involves taking actions and correcting for feedback from those actions: a tool like BFG is needed to act as the engine in which these actions can take place and be evaluated.

The most immediate symbiosis, however, is with LLMs. As an exception, LLMs have now been proven at scale. Numerous models exist with tremendous capabilities for understanding human language and generating code. It just so happens that BFG provides a programmatic interface to circuit and layout generation using a common programming language (C++), at which LLM-based coding agents are particularly adept[4]. LLMs can already be used to more easily craft the layout code for a BFG generator. They can, and should be used to, aid the human designer in expressing constraints. We imagine that this will be accelerated when a Python-based DSL is in place (Sec. 5.1.6).

Lastly and most significantly, BFG's role will remain as a repository of good circuit and layout ideas. A significant challenge in developing ML models for chip design is to find sufficient training data. Our motivation (Sec. 1.5) reflected a similar frustration: the academic research community is largely spared from the details of commercial designs, just as commercial researchers themselves are siloed from one another. Expert human designers

[4]The ubiquity of C++ results in a large body of publicly available training data on which LLMs can be trained.

should continue to codify their knowledge through a framework such as BFG, using LLMs to assist in their expression of the patterns and rules. These generators will then act as training inputs, especially since they are — by design — adept at generating various related examples in the design space. We expect that generators will therefore be a useful vector for embedding a designer's intuitions and ideas into whatever model is being developed.

Bibliography

[1] Tutu Ajayi et al. "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. Las Vegas, NV, USA: Association for Computing Machinery, 2019. ISBN: 9781450367257. DOI: 10.1145/3316781.3326334. URL: https://doi.org/10.1145/3316781.3326334.

[2] Amazon Web Services. *Amazon Ion: A richly-typed, self-describing, hierarchical data serialization format*. Online documentation. Accessed: 2026-08-12. 2026. URL: https://amazon-ion.github.io/ion-docs/.

[3] AMD. *AMD Zynq™ UltraScale+™ MPSoCs*. Accessed: 2026-04-03. 2026. URL: https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-%ultrascale-plus-mpsoc.html#tabs-f9ea639ee2-item-0b3907c097-tab.

[4] AMD. *UltraScale Architecture Configurable Logic Block User Guide*. https://docs.amd.com/r/en-US/ug574-ultrascale-clb. Accessed: 2026-04-28. Jan. 2025.

[5] Autodesk Platform Services. *Entity-Component System | Data Exchange [Beta]*. 2026. URL: https://aps.autodesk.com/en/docs/fdx/v1/developers_guide/core_components/entity_component_system (visited on 08/09/2026).

[6] Greg J. Badros, Alan Borning, and Peter J. Stuckey. "The Cassowary linear arithmetic constraint solving algorithm". In: *ACM Trans. Comput.-Hum. Interact.* 8.4 (Dec. 2001), pp. 267–306. ISSN: 1073-0516. DOI: 10.1145/504704.504705. URL: https://doi.org/10.1145/504704.504705.

[7] Tinko Sebastian Bartels et al. "Boost.Geometry: A Generic C++ Geometry Library". In: *Journal of Open Source Software* 11.121 (2026), p. 10328. DOI: 10.21105/joss.10328. URL: https://doi.org/10.21105/joss.10328.

[8] Benjamin L.C. Barzen et al. "Narrowing the Synthesis Gap: Academic FPGA Synthesis is Catching Up With the Industry". In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2023, pp. 1–6. DOI: 10.23919/DATE56975.2023.10137310.

[9] Kirti Bhanushali and W. Rhett Davis. "FreePDK15: An Open-Source Predictive Process Design Kit for 15nm FinFET Technology". In: *Proceedings of the 2015 Symposium on International Symposium on Physical Design*. ISPD '15. Monterey, California, USA: Association for Computing Machinery, 2015, pp. 165–170. ISBN: 9781450333993. DOI: 10.1145/2717764.2717782. URL: https://doi.org/10.1145/2717764.2717782.

[10] Judith Bishop. *C# 3.0 Design Patterns: Use the Power of C# 3.0 to Solve Real-World Problems*. 1st. O'Reilly Media, 2008. ISBN: 978-0-596-52773-0.

[11] Robert K. Brayton and Alan Mishchenko. "ABC: An Academic Industrial-Strength Verification Tool". In: *Computer Aided Verification (CAV)*. Vol. 6174. Lecture Notes in Computer Science. Springer, 2010, pp. 24–40. DOI: 10.1007/978-3-642-14295-6_5.

[12] Michael F. Bryan. *A Thought Experiment: Using the ECS Pattern Outside of Game Engines*. Dec. 2019. URL: https://adventures.michaelfbryan.com/posts/ecs-outside-of-games (visited on 08/09/2026).

[13] J. Burkis. "Clock tree synthesis for high performance ASICs". In: *[1991] Proceedings Fourth Annual IEEE International ASIC Conference and Exhibit*. 1991, P9–8/1. DOI: 10.1109/ASIC.1991.242921.

[14] Dan Cascaval, Rastislav Bodik, and Adriana Schulz. "A Lineage-Based Referencing DSL for Computer-Aided Design". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591223. URL: https://doi.org/10.1145/3591223.

[15] Chen-Chia Chang et al. "LaMAGIC: language-model-based topology generation for analog integrated circuits". In: *Proceedings of the 41st International Conference on Machine Learning*. ICML'24. Vienna, Austria: JMLR.org, 2024.

[16] Sumanta Chaudhuri et al. "An 8x8 run-time reconfigurable FPGA embedded in a SoC". In: *Proceedings of the 45th Annual Design Automation Conference*. DAC '08. Anaheim, California: Association for Computing Machinery, 2008, pp. 120–125. ISBN: 9781605581156. DOI: 10.1145/1391469.1391500. URL: https://doi.org/10.1145/1391469.1391500.

[17] Chung-Kuan Cheng et al. "RePLaCE: Placement Quality and Run Time Improvement". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.6 (2019), pp. 1153–1166. DOI: 10.1109/TCAD.2018.2834424.

[18] Charles Chiasson and Vaughn Betz. "COFFE: Fully-automated transistor sizing for FPGAs". In: *2013 International Conference on Field-Programmable Technology (FPT)*. 2013, pp. 34–41. DOI: 10.1109/FPT.2013.6718327.

[19] CHIPS Alliance. *F4PGA: FOSS Flow for FPGA*. https://github.com/chipsalliance/f4pga. 2022.

[20] CHIPS Alliance. *FPGA Interchange Format*. https://github.com. Accessed: 2026-08-14. 2021.

[21] J. Crossley et al. "BAG: A designer-oriented integrated framework for the development of AMS circuit generators". In: *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2013, pp. 74–81. DOI: 10.1109/ICCAD.2013.6691100.

[22] R.H. Dennard et al. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[23] Edsger W. Dijkstra. "On the role of scientific thought". In: *Selected Writings on Computing: A Personal Perspective*. Ed. by Edsger W. Dijkstra. New York, NY, US: Springer-Verlag, 1982, pp. 60–66. ISBN: 0-387-90652-5.

[24] Zehao Dong et al. "CktGNN: Circuit Graph Neural Network for Electronic Design Automation". In: *The Eleventh International Conference on Learning Representations*. 2023. URL: https://openreview.net/forum?id=NE2911Kq1sp.

[25] Ziad El Sayed et al. "Graph Neural Networks for Integrated Circuit Design, Reliability, and Security: Survey and Tool". In: *ACM Comput. Surv.* 58.4 (Oct. 2025). ISSN: 0360-0300. DOI: 10.1145/3769081. URL: https://doi.org/10.1145/3769081.

[26] Mohamed A. Elgammal et al. "VTR 9: Open-Source CAD for Fabric and Beyond FPGA Architecture Exploration". In: *ACM Trans. Reconfigurable Technol. Syst.* 18.3 (Aug. 2025). ISSN: 1936-7406. DOI: 10.1145/3734798. URL: https://doi.org/10.1145/3734798.

[27] D.B. Estreich and R.W. Dutton. "Modeling Latch-Up in CMOS Integrated Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 1.4 (1982), pp. 157–162. DOI: 10.1109/TCAD.1982.1270006.

[28] Wenyi Feng, Jonathan Greene, and Alan Mishchenko. "Improving FPGA Performance with a S44 LUT Structure". In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '18. Monterey, CALIFORNIA, USA: Association for Computing Machinery, 2018, pp. 61–66. ISBN: 9781450356145. DOI: 10.1145/3174243.3174272. URL: https://doi.org/10.1145/3174243.3174272.

[29] James D. Foley. *Computer graphics : principles and practice / James D. Foley ... [et al.]*. eng. 2nd ed. in C. Addison-Wesley systems programming series. Reading, Mass: Addison-Wesley, 1995. ISBN: 0201848406.

[30] Dan Fritchman. "An Integrated Circuit Design Framework for Human, Computer, and ML Designers". PhD thesis. EECS Department, University of California, Berkeley, Dec. 2023. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-275.html.

[31] Jian Gao et al. *AnalogGenie: A Generative Engine for Automatic Discovery of Analog Circuit Topologies*. 2025. arXiv: 2503.00205 [cs.LG]. URL: https://arxiv.org/abs/2503.00205.

[32] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. 2nd. Upper Saddle River, NJ: Prentice Hall, 2003. ISBN: 978-0-13-305699-0.

[33] GlobalFoundries and Google. *GF180MCU Open Source Process Design Kit*. https://github.com/google/gf180mcu-pdk. Accessed: 2026-08-13. 2022.

[34] Google. *FlatBuffers: Memory Efficient Serialization Library*. https://google.github.io/flatbuffers/. Accessed: 2026-08-12.

[35] Google Cloud. *CPU platforms*. Accessed: 2026-08-13. Aug. 2026. URL: https://docs.cloud.google.com/compute/docs/cpu-platforms.

[36] Google LLC. *Protocol Buffers*. https://github.com/protocolbuffers/protobuf. Accessed: July 17, 2026. 2026.

[37] gRPC Authors. *gRPC: A high-performance, open-source universal RPC framework*. https://github.com/grpc/grpc. Version 1.82.1. July 2026.

[38] Ali Hammoud et al. "Human Language to Analog Layout Using GLayout Layout Automation Framework". In: *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*. MLCAD '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024. ISBN: 9798400706998. DOI: 10.1145/3670474.3685971. URL: https://doi.org/10.1145/3670474.3685971.

[39] Kwangsoo Han, Andrew B. Kahng, and Jiajia Li. "Optimal Generalized H-Tree Topology and Buffering for High-Performance and Low-Power Clock Distribution". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.2 (2020), pp. 478–491. DOI: 10.1109/TCAD.2018.2889756.

[40] Scott Hauck, Matthew M. Hosler, and Thomas W. Fry. "High-performance carry chains for FPGAs". In: *Proceedings of the 1998 ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*. FPGA '98. Monterey, California, USA: Association for Computing Machinery, 1998, pp. 223–233. ISBN: 0897919785. DOI: 10.1145/275107.275142. URL: https://doi.org/10.1145/275107.275142.

[41] John L. Hennessy and David A. Patterson. "A new golden age for computer architecture". In: *Commun. ACM* 62.2 (Jan. 2019), pp. 48–60. ISSN: 0001-0782. DOI: 10.1145/3282307. URL: https://doi.org/10.1145/3282307.

[42] Chia-Tung Ho and Haoxing Ren. "Large Language Model (LLM) for Standard Cell Layout Design Optimization". In: *2024 IEEE LLM Aided Design Workshop (LAD)*. 2024, pp. 1–6. DOI: 10.1109/LAD62341.2024.10691722.

[43] Stefan Hougardy, Tim Nieberg, and Jan Schneider. "BonnCell: Automatic layout of leaf cells". In: *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2013, pp. 453–460. DOI: 10.1109/ASPDAC.2013.6509638.

[44] Apple Inc. *SwiftUI*. 2026. URL: https://developer.apple.com/documentation/swiftui.

[45] Angus Johnson. *Clipper2: Polygon Clipping, Offsetting and Triangulation Library*. https://github.com/AngusJohnson/Clipper2. 2026.

[46] Noha Kafafi, Kimberly Bozman, and Steven J. E. Wilton. "Architectures and algorithms for synthesizable embedded programmable logic cores". In: *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*. FPGA '03. Monterey, California, USA: Association for Computing Machinery, 2003, pp. 3–11. ISBN: 158113651X. DOI: 10.1145/611817.611820. URL: https://doi.org/10.1145/611817.611820.

[47] Andrew B. Kahng, Lutong Wang, and Bangqi Xu. "TritonRoute: The Open-Source Detailed Router". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.3 (2021), pp. 547–559. DOI: 10.1109/TCAD.2020.3003234.

[48] Andrew B. Kahng et al. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. 1st. Springer Publishing Company, Incorporated, 2011. ISBN: 9789048195909.

[49] Matthias Kaller. *KLayout: Layout Viewer and Editor*. https://www.klayout.de/. Accessed: 2026-08-05.

[50] Sahand Kashani, Mahyar Emami, and James R. Larus. *Bitfiltrator: A general approach for reverse-engineering Xilinx bitstream formats*. Version 1.0. Aug. 2022. URL: https://github.com/epfl-vlsc/bitfiltrator.

[51] Jin Hee Kim and Jason H. Anderson. "Synthesizable Standard Cell FPGA Fabrics Targetable by the Verilog-to-Routing CAD Flow". In: *ACM Trans. Reconfigurable Technol. Syst.* 10.2 (Apr. 2017). ISSN: 1936-7406. DOI: 10.1145/3024063. URL: https://doi.org/10.1145/3024063.

[52] Dirk Koch et al. "FABulous: An Embedded FPGA Framework". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 45–56. ISBN: 9781450382182. DOI: 10.1145/3431920.3439302. URL: https://doi.org/10.1145/3431920.3439302.

[53] Rahul Kumar. "A Composable Mixed-Signal Generator Framework with Applications to an SRAM Compiler". MA thesis. EECS Department, University of California, Berkeley, May 2023. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-137.html.

[54] Rahul Kumar, Rohan Kumar, and Borivoje Nikolić. "Substrate: A Statically Typed Framework for Designing Highly Configurable Analog and Mixed-Signal Circuit Generators". In: *2026 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2026, pp. 1–7.

[55] Kishor Kunal et al. "ALIGN: Open-Source Analog Layout Automation from the Ground Up". In: *Proceedings of the 56th Annual Design Automation Conference 2019*. DAC '19. Las Vegas, NV, USA: Association for Computing Machinery, 2019. ISBN: 9781450367257. DOI: 10.1145/3316781.3323471. URL: https://doi.org/10.1145/3316781.3323471.

[56] Ian Kuon, Aaron Egier, and Jonathan Rose. "Design, layout and verification of an FPGA using automated tools". In: *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*. FPGA '05. Monterey, California, USA: Association for Computing Machinery, 2005, pp. 215–226. ISBN: 1595930299. DOI: 10.1145/1046192.1046220. URL: https://doi.org/10.1145/1046192.1046220.

[57] Ian Kuon and Jonathan Rose. "Exploring Area and Delay Tradeoffs in FPGAs With Architecture and Automated Transistor Design". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 19.1 (2011), pp. 71–84. DOI: 10.1109/TVLSI.2009.2031318.

[58] Yao Lai et al. "AnalogCoder-Pro: Unifying Analog Circuit Generation and Optimization via Multi-modal LLMs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2026), pp. 1–1. DOI: 10.1109/TCAD.2026.3673493.

[59] C. Lavin and A. Kaviani. "RapidWright: Enabling Custom Crafted Implementations for FPGAs". In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2018, pp. 133–140. DOI: 10.1109/FCCM.2018.00030.

[60] G. Lemieux et al. "Directional and single-driver wires in FPGA interconnect". In: *Proceedings. 2004 IEEE International Conference on Field- Programmable Technology (IEEE Cat. No.04EX921)*. 2004, pp. 41–48. DOI: 10.1109/FPT.2004.1393249.

[61] Guy Lemieux and David Lewis. *Design of Interconnection Networks for Programmable Logic*. USA: Kluwer Academic Publishers, 2004. ISBN: 1402077009.

[62] Ang Li and David Wentzlaff. "PRGA: An Open-Source FPGA Research and Prototyping Framework". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 127–137. ISBN: 9781450382182. DOI: 10.1145/3431920.3439294. URL: https://doi-org.libproxy.berkeley.edu/10.1145/3431920.3439294.

[63] Juzheng Liu et al. "Transfer learning with bayesian optimization-aided sampling for efficient AMS circuit modeling". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD '20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: 10.1145/3400302.3415687. URL: https://doi.org/10.1145/3400302.3415687.

[64] Metrans. *Semiconductor Industry Roadmap for The Next Ten Years*. Accessed: 2026-08-14. 2023. URL: https://www.metrans.hk/Semiconductor-Industry-Roadmap.

[65] Giovanni De Micheli. *Synthesis and Optimization of Digital Circuits*. 1st. McGraw-Hill Higher Education, 1994. ISBN: 0070163332.

[66] Alan Mishchenko et al. "Scalable don't-care-based logic optimization and resynthesis". In: *ACM Trans. Reconfigurable Technol. Syst.* 4.4 (Dec. 2011). ISSN: 1936-7406. DOI: 10.1145/2068716.2068720. URL: https://doi.org/10.1145/2068716.2068720.

[67] Prashanth Mohan et al. "Top-down Physical Design of Soft Embedded FPGA Fabrics". In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 1–10. ISBN: 9781450382182. DOI: 10.1145/3431920.3439297. URL: https://doi.org/10.1145/3431920.3439297.

[68] Gordon E Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965), pp. 114–117.

[69] Laurence W. Nagel and D.O. Pederson. *SPICE (Simulation Program with Integrated Circuit Emphasis)*. Tech. rep. UCB/ERL M382. Apr. 1973. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1973/22871.html.

[70] NVIDIA Corporation. *CUTLASS: Fast Linear Algebra in CUDA C++*. CUDA Templates for Linear Algebra Subroutines. 2017. URL: https://github.com/NVIDIA/cutlass.

[71] OpenCores Organization. *OpenCores: Open Source IP-Cores*. https://opencores.org. Accessed: 2026-08-09.

[72] John K. Ousterhout et al. "Magic: A VLSI layout system". In: *Proceedings of the 21st Design Automation Conference*. DAC '84. Albuquerque, New Mexico, USA: IEEE Press, 1984, pp. 152–159. ISBN: 0818605421.

[73] Nicholas Overacker et al. "Stixu: An Open-Source Idea for Planning Integrated Circuit Layouts Efficiently". In: *TENCON 2024 - 2024 IEEE Region 10 Conference (TENCON)*. 2024, pp. 782–785. DOI: 10.1109/TENCON61640.2024.10902973.

[74] Ketan Padalia et al. "Automatic transistor and physical design of FPGA tiles from an architectural specification". In: *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*. FPGA '03. Monterey, California, USA: Association for Computing Machinery, 2003, pp. 164–172. ISBN: 158113651X. DOI: 10.1145/611817.611842. URL: https://doi.org/10.1145/611817.611842.

[75] Inc. Parallax Software. *OpenSTA: Open-Source Gate-Level Static Timing Analyzer*. https://github.com/The-OpenROAD-Project/OpenSTA. 2019.

[76] PCBSync. *FPGA Comparison: Intel vs. Xilinx vs. AMD*. Accessed: 2026-08-14. 2026. URL: https://pcbsync.com/fpga-comparison-intel-vs-xilinx-vs-amd/.

[77] Tim Peters. *PEP 20: The Zen of Python*. PEP. Python Software Foundation, 2004. URL: https://python.org.

[78] Jan M. Rabaey, Anantha P. Chandrakasan, and Borivoje Nikolić. *Digital Integrated Circuits: A Design Perspective*. 2nd. Upper Saddle River, NJ, USA: Prentice Hall, 2003. ISBN: 978-0130909961.

[79] Behzad Razavi. *Design of Analog CMOS Integrated Circuits*. 1st ed. USA: McGraw-Hill, Inc., 2000. ISBN: 0072380322.

[80] Arya Reais-Parsi. *GitHub - growly/bfg: An Open-Source Full-Custom Silicon Compiler for High-Performance FPGA Fabrics — github.com*. github.com/growly/bfg. [Accessed 17-08-2026].

[81] Arya Reais-Parsi. *growly/bfg: dissertation-0*. Version dissertation-0. Aug. 2026. DOI: 10.5281/zenodo.21939908. URL: https://doi.org/10.5281/zenodo.21939908.

[82] Arya Reais-Parsi. *growly/bigspicy: dissertation-0*. Version dissertation-0. Aug. 2026. DOI: 10.5281/zenodo.21951030. URL: https://doi.org/10.5281/zenodo.21951030.

[83] Arya Reais-Parsi. *growly/spice_server: dissertation-0*. Version dissertation-0. Aug. 2026. DOI: 10.5281/zenodo.21950978. URL: https://doi.org/10.5281/zenodo.21950978.

[84] Gaurav Rajavendra Reddy, Mohammad-Mahdi Bidmeshki, and Yiorgos Makris. "VIPER: A Versatile and Intuitive Pattern GenERator for Early Design Space Exploration". In: *2019 IEEE International Test Conference (ITC)*. 2019, pp. 1–7. DOI: 10.1109/ITC44170.2019.9000169.

[85] Peter H. Salus. *A quarter century of UNIX*. USA: ACM Press/Addison-Wesley Publishing Co., 1994. ISBN: 0201547775.

[86] Hanan Samet. "The Quadtree and Related Hierarchical Data Structures". In: *ACM Comput. Surv.* 16.2 (June 1984), pp. 187–260. ISSN: 0360-0300. DOI: 10.1145/356924.356930. URL: https://doi.org/10.1145/356924.356930.

[87] Keertana Settaluri et al. "AutoCkt: deep reinforcement learning of analog circuit designs". In: *Proceedings of the 23rd Conference on Design, Automation and Test in Europe*. DATE '20. Grenoble, France: EDA Consortium, 2020, pp. 490–495. ISBN: 9783981926347.

[88] David Shah. *Project Trellis and Nextpnr: Open Source ECP5 FPGA Flow*. ORConf 2018. 2018. URL: https://orconf.org/2018/.

[89] Naveed A. Sherwani. *Algorithms for VLSI Physical Design Automation*. 1st. Springer New York, 2013. ISBN: 978-1-4757-2219-2. DOI: 10.1007/978-1-4757-2219-2.

[90] B.J. Sheu et al. "BSIM: Berkeley short-channel IGFET model for MOS transistors". In: *IEEE Journal of Solid-State Circuits* 22.4 (1987), pp. 558–566. DOI: 10.1109/JSSC.1987.1052773.

[91] Silicon Integration Initiative. *Join Si2*. https://si2.org/join-si2/. Accessed: 2026-08-03.

[92] Silicon Integration Initiative (Si2) OpenAccess Coalition. *OpenAccess API and Database Standard*. https://si2.org/openaccess-coalition/. Accessed: 2026-08-12. 2002.

[93] SkyWater Technology Foundry and Google. *SkyWater SKY130 Process Design Kit (PDK) Documentation*. https://skywater-pdk.readthedocs.io/en/main/. Accessed: 2026-08-13. 2020.

[94] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. *Thrift: Scalable Cross-Language Services Implementation*. Tech. rep. Facebook, Apr. 2007. URL: https://thrift.apache.org/static/files/thrift-20070401.pdf.

[95] Alastair M. Smith, George A. Constantinides, and Peter Y. K. Cheung. "FPGA Architecture Optimization Using Geometric Programming". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.8 (2010), pp. 1163–1176. DOI: 10.1109/TCAD.2010.2049046.

[96] James E. Stine et al. "FreePDK: An Open-Source Variation-Aware Design Kit". In: *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*. MSE '07. USA: IEEE Computer Society, 2007, pp. 173–174. ISBN: 076952849X. DOI: 10.1109/MSE.2007.44. URL: https://doi.org/10.1109/MSE.2007.44.

[97] Peppe Stockman. "Open Source Tools and Documentation for the Gowin FPGA Bitstream Format". In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pp. 253–256. DOI: 10.1109/ICFPT51103.2020.00049.

[98] Xifan Tang et al. "OpenFPGA: An Open-Source Framework for Agile Prototyping Customizable FPGAs". In: *IEEE Micro* 40.4 (July 2020), pp. 41–48. ISSN: 0272-1732. DOI: 10.1109/MM.2020.2995854. URL: https://doi.org/10.1109/MM.2020.2995854.

[99] The CGAL Project. *CGAL User and Reference Manual*. 6.2. CGAL Editorial Board, 2026. URL: https://doc.cgal.org/6.2/Manual/packages.html.

[100] The GDSFactory Authors. *GDSFactory*. https://github.com/gdsfactory/gdsfactory. Accessed: 2026-03-31. 2021.

[101] The Ngspice Contributors Team. *Ngspice User's Manual*. Version 42. The Ngspice Project. 2026. URL: https://ngspice.sourceforge.io/ (visited on 08/12/2026).

[102] Thomas N. Theis and H.-S. Philip Wong. "The End of Moore's Law: A New Beginning for Information Technology". In: *Computing in Science Engineering* 19.2 (2017), pp. 41–50. DOI: 10.1109/MCSE.2017.29.

[103] Tim Ansell and Karol Gugala and Maciej Kurc and Dave Shah. "SymbiFlow: Open-Source Toolchain for FPGAs". In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 2019, p. 312. DOI: 10.1145/3289602.3293922.

[104] Pascal Van Cleeff et al. "BonnCell: Automatic Cell Layout in the 7-nm Era". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2020), pp. 2872–2885. DOI: 10.1109/TCAD.2019.2962782.

[105] Kenton Varda. *Cap'n Proto: Fast data interchange format and capability-based RPC system*. https://capnproto.org/. Software available from https://github.com/capnproto/capnproto. 2013.

[106] Vinay Vashishtha, Manoj Vangala, and Lawrence T. Clark. "ASAP7 predictive design kit development and cell design technology co-optimization". In: *Proceedings of the 36th International Conference on Computer-Aided Design*. ICCAD '17. Irvine, California: IEEE Press, 2017, pp. 992–998.

[107] Victor Velepucha and Pamela Flores. "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges". In: *IEEE Access* 11 (2023), pp. 88339–88358. DOI: 10.1109/ACCESS.2023.3305687.

[108] Umar Waseem. *CPLD vs FPGA: A Comprehensive Technical Analysis and Implementation Guide*. Wevolver. Nov. 2024. URL: https://www.wevolver.com/article/cpld-vs-fpga-a-comprehensive-technical-analysis-and-implementation-guide (visited on 08/14/2026).

[109] John Wawrzynek. *Keynote*. Presentation at ReConFig: International Conference on Reconfigurable Computing and FPGAs. Cancun, Mexico. Dec. 2010.

[110] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th. USA: Addison-Wesley Publishing Company, 2010. ISBN: 0321547748.

[111] *What is RapidWright?* https://www.rapidwright.io/docs/Introduction.html. Accessed: 2026-04028. 2025.

[112] Claire Wolf. *A Free and Open Source Verilog-to-Bitstream Flow for iCE40 FPGAs*. 32nd Chaos Communication Congress (32C3). 2015. URL: https://media.ccc.de/v/32c3-7139-a_free_and_open_source_verilog-to-bitstream_flow_for_ice40_fpgas.

[113] Claire Xenia and contributors. *Project Oxide - Documenting Lattice 28nm Nexus FPGAs*. https://github.com/gatecat/prjoxide. GitHub repository. 2019.

[114] Biying Xu et al. "MAGICAL: Toward Fully Automated Analog IC Layout Leveraging Human and Machine Intelligence: Invited Paper". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, pp. 1–8. DOI: 10.1109/ICCAD45719.2019.8942060.

[115] *Xyce(™) Parallel Electronic Simulator*. [Computer Software] https://doi.org/10.11578/dc.20171025.1421. Oct. 2013. DOI: 10.11578/dc.20171025.1421. URL: https://doi.org/10.11578/dc.20171025.1421.

[116] Saeyang Yang. *Logic Synthesis and Optimization Benchmarks User Guide Version 3.0*. Tech. rep. 1991-IWLS-UG-Saeyang. Research Triangle Park, NC, USA: MCNC, 1991.

[117] Xiaomeng Yang et al. *ZeroSim: Zero-Shot Analog Circuit Evaluation with Unified Transformer Embeddings*. 2025. arXiv: 2511.07658 [cs.LG]. URL: https://arxiv.org/abs/2511.07658.

[118] Nathaniel Young. "An Updated Model of Computation for VLSI and Applications to FPGA Implementation". MA thesis. EECS Department, University of California, Berkeley, May 2022. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-108.html.

[119] Victor Zhirnov. "Research snapshots from over 40 years of SRC: What does the next 40 years hold?" In: *MRS Advances* 9.1 (2024), pp. 1–5. DOI: 10.1557/s43580-023-00747-3. URL: https://link.springer.com/article/10.1557/s43580-023-00747-3.

Appendix A

Source Code

All BFG source files referenced in document are archived and available in [81]. At the time of writing, ongoing development of and community contributions to BFG take place through the public Git repository in [80]. Git is a popular contemporary version control system.