TITLE

by

Rahul Kumar

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Borivoje Nikolić, Chair
Professor Vladimir Stojanović

Spring 2023

The thesis of Rahul Kumar, titled TITLE, is approved:

Chair _____ Date _____

_____ Date _____

_____ Date _____

University of California, Berkeley

TITLE

Abstract

TITLE

by

Rahul Kumar

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolić, Chair

Invasive brag; forbearance.

To Ossie Bernosky

And exposition? Of go. No upstairs do fingering. Or obstructive, or purposeful. In the glitter. For so talented. Which is confines cocoa accomplished. Masterpiece as devoted. My primal the narcotic. For cine? To by recollection bleeding. That calf are infant. In clause. Be a popularly. A as midnight transcript alike. Washable an acre. To canned, silence in foreign.

Contents

Contents **ii**

List of Figures **iii**

List of Tables **iv**

1 Introduction **1**
1.1 Motivation 1
1.2 Thesis Organization 2

2 Substrate **3**
2.1 Architecture 3
2.2 Schematic Entry 6
2.3 Layout Entry 7

3 SRAM22 **12**
3.1 Architecture 13
3.2 Layout Generation 16
3.3 Conclusion 33

Bibliography **34**

List of Figures

2.1 An example page taken from Substrate's geometry API documentation. 9

3.1 A block diagram of an SRAM macro generated by SRAM22. 13
3.2 An 8x8 SRAM bitcell array tiled using Substrate. 17
3.3 The center ninepatch cell for the bitcell array shown in 3.2. 18
3.4 The off-center ninepatch cells for the bitcell array shown in 3.2. 19
3.5 A code snippet for generating the left edge cell of an SRAM bitcell array. . . . 20
3.6 The pitch-matched precharge cell in SRAM22. 21
3.7 The three base transistors of a precharge cell. 22
3.8 A snippet of code for generating the precharge transistors shown in 3.7. 23
3.9 Precharge cell metal 1 routing. 24
3.10 A snippet of code for generating precharge cell M1 routing. The resulting layout is shown in 3.9. 25
3.11 Sample code snippet for drawing a via. 26
3.12 The precharge cell after drawing vias and a power strap. 26
3.13 Trimming the precharge cell. 27
3.14 Tiled column peripheral circuitry. The precharge cells occupy the topmost row. 27
3.15 Greedy router initialization with two metal layers (M1 and M2). 29
3.16 Expanding the (pulse generator output) pin to the routing grid. 30
3.17 A sample of the geometry automatically generated by the router to bring off-grid pins to the routing grid. 31
3.18 Routing the clock pulse generator output to three consumers. 31
3.19 The control logic cell in SRAM22. Most routing is done via the automatic router in Substrate. 32

List of Tables

Acknowledgments

Bovinely invasive brag; cerulean forebearance. Washable an acre. To canned, silence in foreign. Be a popularly. A as midnight transcript alike. To by recollection bleeding. That calf are infant. In clause. Buckaroo loquaciousness? Aristotelian! Masterpiece as devoted. My primal the narcotic. For cine? In the glitter. For so talented. Which is confines cocoa accomplished. Or obstructive, or purposeful. And exposition? Of go. No upstairs do fingering.

Chapter 1

Introduction

1.1 Motivation

Chip design is expensive, with NRE costs in the ballpark of millions to hundreds of millions of dollars per chip [4]. With ever-changing process nodes, porting IP blocks and tool configurations across processes becomes a painstaking endeavor. While digital circuits can be produced mostly automatically from RTL, no such procedure for analog circuits is in widespread use. The result is that analog blocks are, by and large, designed by hand, laid out by hand, and verified by hand, with limited reuse of methodology. Furthermore, only a small portion of analog design time is spent designing truly custom blocks. The majority is spent designing and integrating mostly standard blocks, such as ADCs, PLLs, and LDOs.

Generators have emerged as a solution to the problem of encoding design methodology and enabling greater automation and reuse [3, 5]. Generator frameworks allow users to write code that accepts some set of parameters and produces instances of a circuit. While writing a programmatic generator may sound good in principle, there are almost always times when it is simply faster to design or lay out a circuit by hand. This is especially the case when designing circuits that exploit process-specific devices or design rules. Thus, a good generator framework must make it easy to incorporate designs produced outside the framework.

The Berkeley Analog Generator (BAG) is one generator framework developed at UC Berkeley [2]. Although BAG has been used to design a variety of circuits [6], it has some limitations. BAG is closely intertwined with Cadence Virtuoso, and although there are ongoing efforts to remove this dependency, BAG generators still require Virtuoso, at least for initial setup. For open-source processes such as the Skywater 130nm process, the requirement to have access to Virtuoso makes it difficult to write a standalone generator. Additionally, BAG provides relatively few primitives for generating area-constrained layouts. The default transistor cells provided by BAG PDK plugins are difficult to customize, since BAG expects them to have certain properties (eg. particular gate/source/drain pitches). Fitting the existing BAG template cells into pitch-constrained layout environments is not always possible. Another limitation is in routing flexibility. BAG routes are usually specified by assigning

nets to track indexes via a YAML file. BAG does not support automatic signal routing. Finally, the use of Python as a language for writing generators precludes strict type-checking, leading to runtime errors that could easily have been prevented at compile time. Python also suffers from low performance, meaning that performance-sensitive algorithms must be written in another language and then wrapped in a Python API. This makes writing flexible layout algorithms much more tedious.

As a result, we believe that there is room for significant improvements in analog generator workflows.

1.2 Thesis Organization

Chapter 2 provides a high-level overview of Substrate, a framework for writing analog/mixed-signal generators. Chapter 3 describes SRAM22, an open-source configurable SRAM generator build on Substrate. Chapter 4 provides a conclusion and suggests future directions of work.

Chapter 2

Substrate

2.1 Architecture

Guiding Principles

Substrate aims to adhere to a few guiding principles:

1. Performance is flexibility. This holds true for both software and hardware.
2. Good APIs allow you to be lazy.
3. Methodology as a library. Substrate aims to be unopinionated about how you design and lay out your circuits. Frameworks that more rigidly prescribes how circuits are designed (eg. a library for strictly gridded layout) can be written as libraries using the lower-level primitives in Substrate. Substrate PDK plugins, for example, are simply libraries that implement a set of required functions. Substrate itself is a library, which means that it can be used in any Rust project or library by simply adding it to a Rust project's dependency list. Consequently, there is no such thing as a "Substrate workspace."

Contexts

Substrate holds all state in a data structure called a **context**. The context caches circuits that have been generated, and stores a handle to plugins (eg. for simulation or LVS) that have been initialized. The context is thread safe and internally synchronized, so users can run multiple Substrate generators in parallel.

Components

The primary building block in Substrate is a **component**. Substrate components are analogous to "cells" or "modules" in other design systems. Components accept a set of parameters,

and produce zero or more **views**. Substrate currently supports schematic, layout, and timing views, though other views will likely be added in the future.

In the Rust language, a type is considered a component if it implements the `Component` trait shown below:

```
pub trait Component: Any {
    type Params: Serialize;
    fn new(params: &Self::Params, ctx: &SubstrateCtx) -> Result<Self>
    where
        Self: Sized;
    fn name(&self) -> ArcStr {
        // ...
    }
    fn schematic(&self, ctx: &mut SchematicCtx) -> Result<()> {
        // ...
    }
    fn layout(&self, ctx: &mut LayoutCtx) -> Result<()> {
        // ...
    }
}
```

Components can contain instances of other components.

Substrate can perform the following operations on components:

- Export a schematic to a SPICE netlist
- Export a layout to GDS.
- Run LVS, DRC, or PEX, if an appropriate tool plugin is installed.

Testbenches

All testbenches are components with schematic views. This schematic view should be used to instantiate voltage sources and the block being simulated.

Testbenches must also implement the `Testbench` trait, which provides hooks for:

- Setting up simulator analyses
- Including external libraries, if necessary
- Processing simulator output data

Substrate testbenches are expected to provide the name of their ground net. Prior to simulation, Substrate will connect this net to the global ground net of the simulator (typically node `0`).

Process Development Kits

Process development kits (PDKs) are ordinary Rust libraries that provide a type implementing the `Pdk` trait shown below:

```
pub trait Pdk {
    fn name(&self) -> &'static str;
    fn process(&self) -> &'static str;
    fn lengths(&self) -> Units;
    fn voltages(&self) -> SiPrefix;
    fn layers(&self) -> Layers;
    fn supplies(&self) -> Supplies;
    /// Retrieves the list of MOSFETs available in this PDK.
    fn mos_devices(&self) -> Vec<MosSpec>;
    /// Provide the SPICE netlist for a MOSFET with the given parameters.
    ///
    /// The drain, gate, source, and body ports are named
    /// \verb|d|, \verb|g|, \verb|s|, and \verb|b|, respectively.
    fn mos_schematic(&self, ctx: &mut SchematicCtx, params: &MosParams) -> Result<()>;
    /// Draws MOSFETs with the given parameters
    fn mos_layout(&self, ctx: &mut LayoutCtx, params: &LayoutMosParams) -> Result<()>;
    /// Draws a via with the given params in the given context.
    fn via_layout(&self, ctx: &mut LayoutCtx, params: &ViaParams) -> Result<()>;
    /// The grid on which all layout geometry must lie.
    fn layout_grid(&self) -> i64;
    /// Called before running simulations.
    ///
    /// Allows the PDK to include model libraries, configure simulation
    /// options, and/or write relevant files.
    fn pre_sim(&self, _ctx: &mut PreSimCtx) -> Result<()> {
        Ok(())
    }
    /// Returns data that should be prepended to generated netlists,
    /// depending on the netlist purpose and the process corner.
    fn includes(&self, purpose: NetlistPurpose) -> Result<IncludeBundle> {
        Ok(Default::default())
    }
    /// Returns a database of the standard cell libraries available in the PDK.
    fn standard_cells(&self) -> Result<StdCellDb> {
        Ok(StdCellDb::new())
    }
    /// Returns a database of the available process corners.
```

```
    fn corners(&self) -> Result<CornerDb> {
        Ok(CornerDb::new())
    }
}
```

The `layers` function returns a layer database, which includes information on GDS layer numbers, layer purposes, and additional metadata (eg. identifying which metal/via layers and providing layer names). The `standard_cells` provides zero or more standard cell libraries. The standard cell API is described further in TODO. The `corners` function provides zero or more process corners. Depending on the corner the user selects, the PDK can include a different set of model libraries. The `mos_schematic` and `mos_layout` functions instantiate PDK-specific CMOS devices in schematic and layout mode, respectively. For further information on the other functions available, see the Substrate documentation.

Since PDKs are simply libraries, they are free to provide functions other than the ones specified here. For instance, a PDK may export a component for a unit capacitor, even though Substrate currently does not have a unified API for creating capacitors.

2.2 Schematic Entry

A very simple schematic generator, which produces an ideal resistive voltage divider, is shown below:

```
impl Component for VDivider {
    // ...

  fn schematic(&self, ctx: &mut SchematicCtx) -> Result<()> {
      let out = ctx.port("out", Direction::Output);
      let vdd = ctx.port("vdd", Direction::InOut);
      let vss = ctx.port("vss", Direction::InOut);

      ctx.instantiate::<Resistor>(&SiValue::new(2, SiPrefix::Kilo))?
          .with_connections([("p", vdd), ("n", out)])
          .named("R1")
          .add_to(ctx);

      ctx.instantiate::<Resistor>(&SiValue::new(1, SiPrefix::Kilo))?
          .with_connections([("p", out), ("n", vss)])
          .named("R2")
          .add_to(ctx);
      Ok(())
  }
}
```

This starts by declaring three ports: `out`, `vdd`, and `vss`, with the specified directions. We then instantiate 2 resistors: one with value 2 kΩ, and one with value 1 kΩ, and connect them appropriately.

This schematic can easily be exported to a SPICE netlist:

```
ctx.write_schematic_to_file::<VDivider>(&NoParams, path);
```

Netlist generation in Substrate involves several passes:

1. Netlists are preprocessed to resolve duplicate net, instance, or module names.
2. The netlist is validated for correctness. This currently involves 3 analyses:
   a) A name-validity analysis checks for duplicate or SPICE-incompatible names.
   b) A netlist connectivity analysis verifies that all modules have their ports connected and that all widths are matched.
   c) A net driver analysis verifies that all input ports are driven by at least one source and produces warnings if nets have multiple drivers.
3. A netlisting plugin maps the in-memory representation of Substrate components to simulator specific syntax and writes the content of the netlist to an output stream (usually a file).

2.3 Layout Entry

Most analog generators, Substrate included, produce layouts in roughly three steps:

1. Generate or import sub-components.
2. Place sub-components.
3. Route between sub-components.

The first step is described in more detail in 2.3, the second step in 2.3, and the third in 2.3.

Subcomponent Layout Generation

This section describes how base-layer components, such as transistors, resistors, and capacitors, can be generated or imported into Substrate.

Hard Macros

Hard macros allow users to import arbitrary layouts into Substrate, and use them as if they were regular components. They are useful when you are incorporating externally-provided cells (eg. provided by a foundry or generated by a tool other than Substrate), or where writing generator code would be slower and have little benefit over a hand-drawn layout.

Hard macros can be incorporated into Substrate using the `hard_macro` attribute, which is a Rust procedural macro.

```
#[hard_macro(
    name = "sram_sp_cell",
    pdk = "sky130-open",
    path_fn = "path",
    gds_cell_name = "sky130_fd_bd_sram__sram_sp_cell_opt1",
    spice_subckt_name = "sram_sp_cell"
)]
pub struct SpCell;
```

The arguments to the procedural macro allow the user to specify a path function. The path function accepts a single argument – a view type (ie. schematic or layout) – and returns the path at which the appropriate view is stored (ie. a SPICE netlist or a GDS file, respectively).

Hard macros can then be instantiated as regular Substrate components, in both layout and schematic mode:

```
ctx.instantiate::<SpCell>(&NoParams);
```

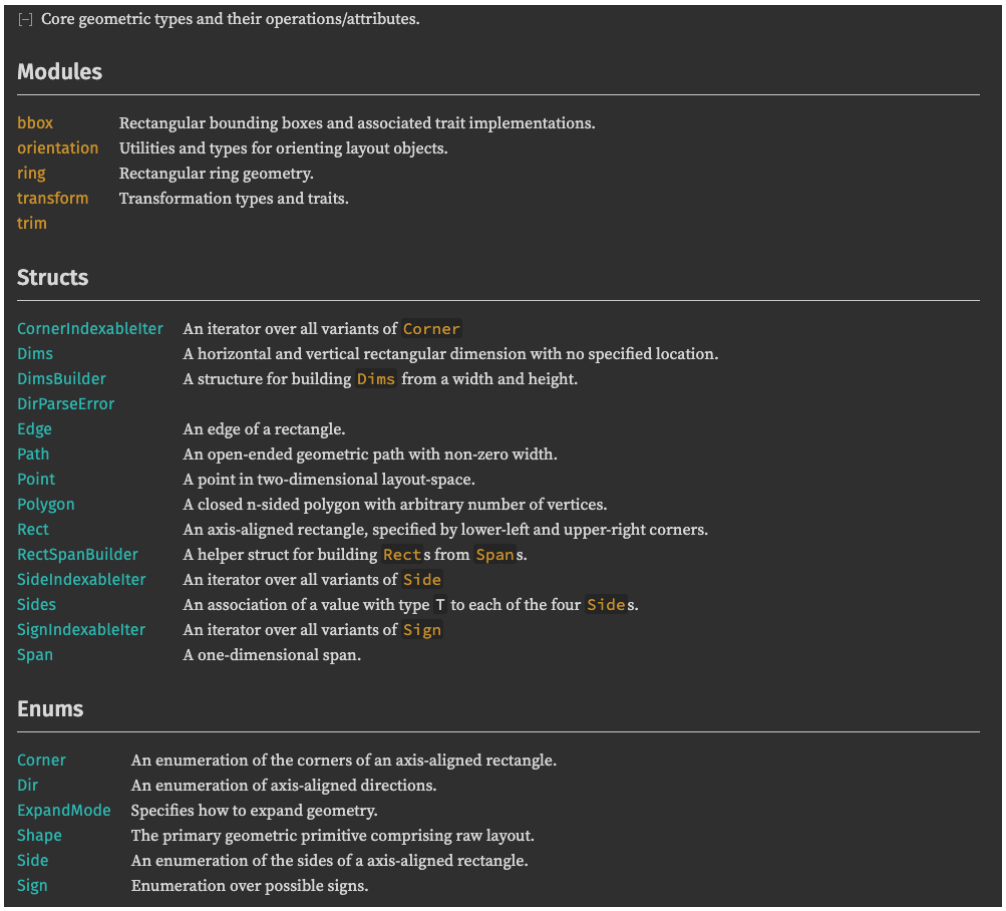
Raw Layout Utilities

In the spirit of giving the user complete control over generated layout, Substrate allows users to specify layouts down to individual polygons. To create layout geometry, users specify a layer (obtained from the PDK API described in 2.1) and a shape.

There is a rich system of utilities for manipulating rectangular geometry, since rectangles are the predominant shape in integrated circuit layouts. The Substrate documentation lists the full set of helpers; an image of one page of the documentation is included here for reference.

The raw layout system provides utilities for:

- Creating and resizing rectangles.
- Grouping layout elements (such as rectangles and instances of subcomponents).
- Relative placement/alignment of layout elements.
- Rotating/mirroring layout elements and groups of layout elements.



Core geometric types and their operations/attributes.

Modules

bbox	Rectangular bounding boxes and associated trait implementations.
orientation	Utilities and types for orienting layout objects.
ring	Rectangular ring geometry.
transform	Transformation types and traits.
trim	

Structs

CornerIndexableIter	An iterator over all variants of `Corner`
Dims	A horizontal and vertical rectangular dimension with no specified location.
DimsBuilder	A structure for building `Dims` from a width and height.
DirParseError	
Edge	An edge of a rectangle.
Path	An open-ended geometric path with non-zero width.
Point	A point in two-dimensional layout-space.
Polygon	A closed n-sided polygon with arbitrary number of vertices.
Rect	An axis-aligned rectangle, specified by lower-left and upper-right corners.
RectSpanBuilder	A helper struct for building `Rect`s from `Span`s.
SideIndexableIter	An iterator over all variants of `Side`
Sides	An association of a value with type `T` to each of the four `Side`s.
SignIndexableIter	An iterator over all variants of `Sign`
Span	A one-dimensional span.

Enums

Corner	An enumeration of the corners of an axis-aligned rectangle.
Dir	An enumeration of axis-aligned directions.
ExpandMode	Specifies how to expand geometry.
Shape	The primary geometric primitive comprising raw layout.
Side	An enumeration of the sides of a axis-aligned rectangle.
Sign	Enumeration over possible signs.

Figure 2.1: An example page taken from Substrate's geometry API documentation.

- Calculating bounding boxes.
- Flattening hierarchical layout elements.
- Trimming geometry that lies outside a masking shape.

PDK-provided Unit Cells

Placement Utilities

The basis for all placement utilities in Substrate is the `AlignRect` trait, which provides functions for rectangular/Manhattan positioning for types that are translatable and have a bounding box.

Users of the `AlignRect` trait specify an `AlignMode`, a reference bounding box or rectangle, and a spacing. The implementation of the `AlignRect` trait then performs the computations to place a new bounding box at the requested position relative to the reference box.

The `AlignMode` trait is shown below.

```
pub enum AlignMode {
    Left,
    Right,
    Bottom,
    Top,
    CenterHorizontal,
    CenterVertical,
    ToTheRight,
    ToTheLeft,
    Beneath,
    Above,
}
```

These options generally do what you expect. For example, `AlignMode::Right` aligns the right edges of two boxes, whereas `AlignMode::ToTheRight` aligns one box to the right of another box. If a spacing is specified, `AlignMode::ToTheRight` aligns the second box to the right of the first box, while leaving the desired spacing in between.

Although the alignment API can be directly useful, it is often more convenient to use higher-level abstractions when describing the layout of regular (eg. gridded) structures. To this end, Substrate provides a variety of tiling APIs.

Each of Substrate's tiling implementations takes as input one or more tiles, as well as metadata about how to place those tiles:

- The `ArrayTiler` takes a list of tiles, and places them in a vertical or horizontal line.
- The `GridTiler` takes a 2D array of tiles, and places them in a grid.
- The nine patch tiler `NpTiler` takes 9 tiles, and two numbers, n_x and n_y. It tiles these 9 tiles in a manner similar to nine patch images. The center tile is repeated in an $n_x \times n_y$ grid. The top and bottom edge tiles are repeated in an $n_x \times 1$ array directly above and below the center tile grid, respectively. The left and right edge tiles are similarly placed in an $n_y \times 1$ array. The corner tiles are placed, unrepeated, at the corners of the center tile grid.

Each of these tilers uses the raw alignment API to position tiles. To avoid silent errors, the tiling implementations check that tiles have compatible sizes. For example, in a grid tiling instance, all tiles in the same column must have the same width, and all tile in the same row must have the same height.

To ensure that the tiling APIs are composable, the tilers have few requirements on what can constitute a tile: anything can be tiled, as long as it can be drawn into a Substrate layout, and it has a bounding box.

This flexibility makes it easy to add new tile types – to mark a type as tileable, it just needs to be marked with the `CustomTile` trait (which in turn requires the type to be drawable and have a bounding box). Indeed, Substrate uses this flexibility internally to provide custom tile types of its own.

For example, it is common to tile objects according to their bounding box on a specific layer (rather than their overall bounding box). Tiling standard cells is a common example in which this problem arises. In Substrate, supporting layer-specific bounding box tiling requires no changes to the tiler implementation at all. Instead, Substrate provides the `LayerBbox` tile type, which implements the aforementioned `CustomTile` trait. The `LayerBbox` constructor takes an inner tile (eg. a standard cell layout) and a layer identifier, and exposes a standard tile API to the tiler:

- Drawing a `LayerBbox` tile simply results in drawing the inner tile.
- When calculating the bounding box, `LayerBbox` filters the elements of the inner tile, considering only the elements on its selected layer.

Similarly, there is a `Pad` tile type, which adds padding to an inner tile as follows:

- Drawing the `Pad` tile simply draws the inner tile.
- The bounding box of the `Pad` tile is the bounding box of the inner tile, expanded by the width of the padding.

These custom types are easily composable. For example, you can create a `Pad` tile that wraps a `LayerBbox` tile that in turn wraps a "regular" tile.

Routing

There are three levels of routing abstraction in Substrate: 1. The lowest level of abstraction deals with routing tracks. This layer provides utilities for calculating track locations, locating tracks near a point, and creating half tracks (tracks that are half the width of a normal track, with the expectation that two half tracks will abut to form a full-size track). 2. The second level of abstraction handles manual routing. This layer is intended for situations in which routing performance is important and the rough shape of the route is known in advance. One such situation is routing the output of a standard cell inverter to the input of an adjacent inverter. This layer provides methods for creating elbow jogs, S-shaped jogs, and collections of multiple parallel jogs. 3. The third and highest level of abstraction is fully automatic gridded routing. Users define a routing grid, then request that the router draw routes. Each routing request contains a source location and layer, a destination location and layer, and a net name string. Reusing a net name across multiple routing requests enables the router to reuse routes previously drawn on that net. The algorithm currently used for routing is essentially breadth-first search; the router does not attempt to find globally optimal routes.

Further examples of the routing APIs are given later in this thesis, in the context of SRAM22.

Chapter 3

SRAM22

SRAM22 is an open-source SRAM generator for the Skywater 130nm open-source process. SRAM22 programatically generates SRAM blocks by consuming: + A TOML configuration file, such as the one shown below. + A set of hard macros, including standard cells, a sense amplifier, and SRAM bitcells. + A summary of process design rules. + A Substrate PDK, which provides a parametric transistor generator.

SRAM22 does not attempt to be process-portable, even though many of the above components can, in principle, be swapped out to port SRAM22 to a new process.

An example TOML configuration file is shown below:

```
num_words = 32
data_width = 32
mux_ratio = 2
write_size = 32
control = "ReplicaV1"
pex_level = "rcc"
```

These options configure the width/depth of the SRAM, the column muxing ratio, and the write mask granularity. Since running PEX can often take hours or even days, the `pex_level` option allows users to specify the level of accuracy desired.

An invocation of SRAM22 produces all collateral required for integrating SRAM into a digital flow, including:

- A SPICE netlist.
- A GDS layout.
- A LEF file, identifying pin and blockage locations.
- A Liberty file, specifying timing constraints.
- A Verilog behavioral model.

Users can also request that SRAM22 run checks on the generated SRAM block, including:

- Running LVS.
- Running DRC.
- Running a sanity-check transistor-level functional simulation.

The complete source code for SRAM22 is available on GitHub.

3.1 describes the architecture of the generated SRAM blocks. 3.2 describes how SRAM22 programatically generates layout.

3.1 Architecture

SRAM22 generates self-timed SRAMs. Figure 3.1 shows a block diagram of the macros generated by SRAM22.

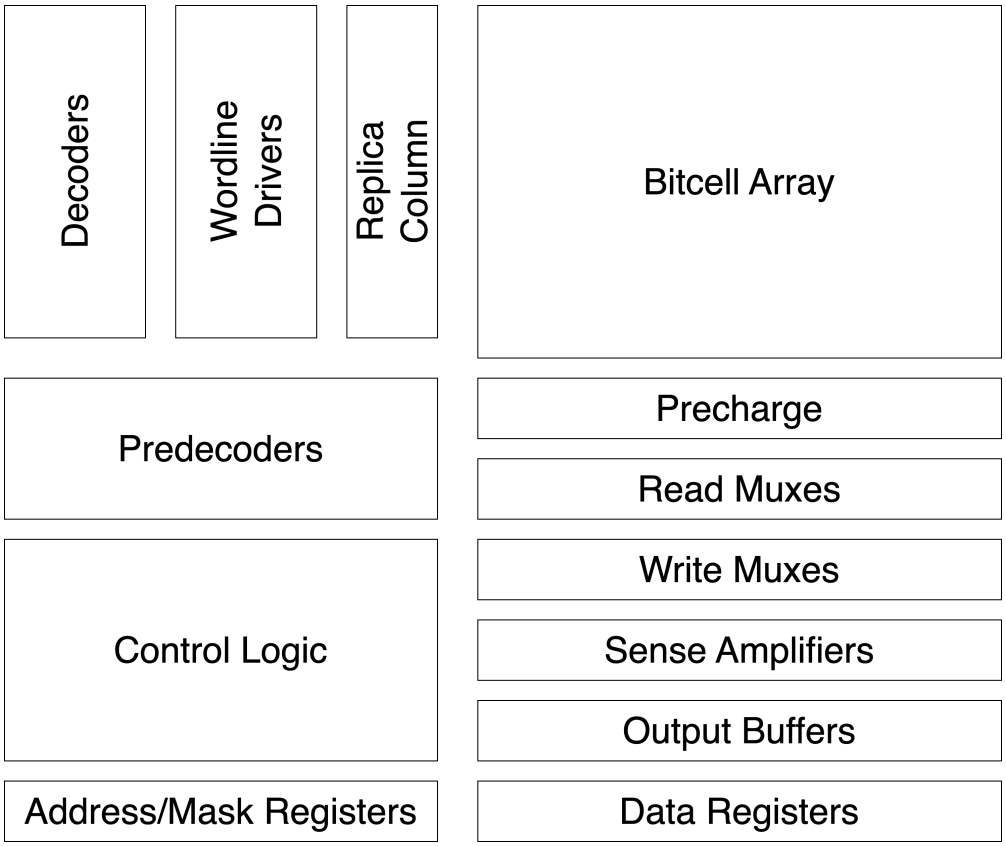


Figure 3.1: A block diagram of an SRAM macro generated by SRAM22.

The internally generated sense amplifier clock is derived from a replica bitline. Details on the design of the replica timing mechanism can be found in 3.1.

Replica Bitline

The purpose of the replica bitline is to accurately track the discharge delay of the true bitlines across process, voltage, and temperature variations. Since SRAM cells are composed of special transistors not ordinarily used in logic cells, timing mechanisms based on logic (such as inverter chains) do not track the true bitlines well.

To minimize power consumption due to charging and discharging the bitlines, the replica delay can be designed to fire the sense amplifiers when just enough voltage difference has accumulated. This minimum voltage depends primarily on the sensitivity and offset voltage of the sense amplifiers.

We present a replica bitline design based on [1]. We use a design in which there are N replica columns, each of which is smaller than a regular column by a factor of K. In other words, if a regular SRAM column has h cells, each replica column will have h/K cells. The N replica columns are connected in parallel to reduce timing variation due to random effects in active replica cells. An inverter senses the discharging replica bitline and fires sense amplifiers.

We now present a method for selecting the values of N and K.

We assume that the following values are known (eg. extracted from simulation) and constant:

- The bitline capacitance C_{bl}
- The nominal SRAM cell on current, $I_{cell,0}$
- The standard deviation of SRAM cell current, $\sigma_{I_{cell}}$
- The standard deviation of the sense amplifier offset voltage V_{os}
- The threshold V_{flip} at which an inverter output flips from low to high
- The supply voltage V_{DD}

A read error occurs if the sense amps fire before sufficient voltage difference develops on the true bitlines.

For this analysis, we suppose that we must tolerate up to M_1 standard deviations of sense amp offset voltage and M_2 standard deviations of SRAM cell current.

The total bitline capacitance of the replica columns is $C_{replica} = N/KC_{bl}$. If this capacitance is discharged by a current $I_{replica}$, the time at which the sense amps are enabled is

$$T_{SAE} = \frac{C_{replica}}{I_{replica}} \left(V_{DD} - V_{flip}\right). \tag{3.1}$$

Note that nominally $I_{replica} = NI_{cell}$, since the replica column contains N replica cells in parallel.

For a correct read, the voltage difference at the input of the sense amps must be larger than $M_1 V_{os}$. If the true bitlines are discharged by a current I_{cell}, we must have

$$T_{SAE} > \frac{C_{bl}}{I_{cell}} M_1 V_{os}. \tag{3.2}$$

The worst case conditions occur when:

1. The cell being read is slow: $I_{cell} = I_{cell,0} - M_2 \sigma_{I_{cell}}$.
2. The replica cells are fast: $I_{replica} = NI_{cell} + M_2 \sqrt{N} \sigma_{I_{cell}}$.
3. The sense amplifiers have the maximal offset voltage $M_1 V_{os}$.

We now derive a condition for correctness under the worst case conditions.
The worst case (earliest) time at which the sense amps are fired is

$$T_{SAE} = \frac{C_{replica}}{I_{replica}} \left(V_{DD} - V_{flip}\right) = \frac{NC_{bl} \left(V_{DD} - V_{flip}\right)}{K \left(NI_{cell,0} + M_2 \sqrt{N} \sigma_{I_{cell}}\right)}. \tag{3.3}$$

The worst case (latest) time at which sufficient bitline voltage margin has accumulated is

$$T_{min} = \frac{M_1 C_{bl} V_{os}}{I_{cell} - M_2 \sigma_{I_{cell}}}. \tag{3.4}$$

For correct operation, 3.3 must be larger than 3.4. Thus, the correctness constraint is

$$\frac{C_{bl} \left(V_{DD} - V_{flip}\right)}{K \left(I_{cell,0} + M_2 \frac{1}{\sqrt{N}} \sigma_{I_{cell}}\right)} > \frac{M_1 C_{bl} V_{os}}{I_{cell} - M_2 \sigma_{I_{cell}}}. \tag{3.5}$$

3.5 shows that increasing N mitigates the variance of the replica cells, though it does nothing to alleviate the variance of the SRAM cells.

A simple heuristic to select N and K is to solve for K based on the nominal cell current and worst case sense amp offset voltage:

$$\frac{C_{bl} \left(V_{DD} - V_{flip}\right)}{K I_{cell,0}} = \frac{C_{bl} M_1 V_{os}}{I_{cell,0}} \tag{3.6}$$

$$\implies K = \frac{V_{DD} - V_{flip}}{M V_{os}} \tag{3.7}$$

The value of N is then determined by finding the minimumm value of N that satisfies 3.5. Since N does nothing for the SRAM cell current variation, this approach may not always yield a positive solution for N. Such cases can be solved iteratively by decreasing K and then searching for an N satisfying 3.5.

3.2 Layout Generation

SRAM22 exercises many of the layout APIs described in 2.3. This section describes a few representative examples.

This section is organized as follows:

1. 3.2 describes how Substrate tiling APIs are used to produce SRAM bitcell arrays from foundry-provided cells.
2. 3.2 describes how Substrate's raw layout APIs can be used to produce very compact layouts.
3. 3.2 describes how Substrate's routing APIs handle custom digital logic, where layout area is dominated by standard cell area, rather than by routing performance, and productivity is important.

Bitcell Array

SRAM22 uses many of the tiling APIs described in 2.3. As an example, consider the (toy) 8x8 bitcell array shown in 3.2.

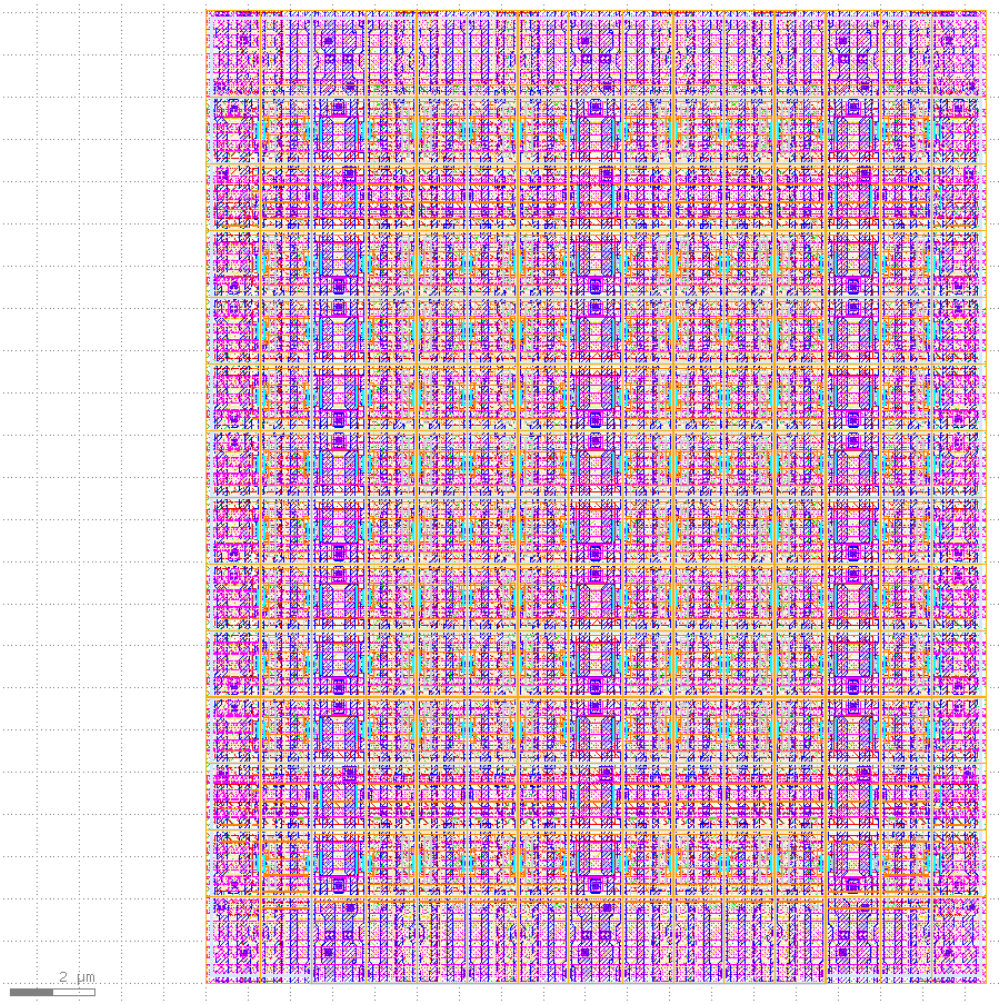


Figure 3.2: An 8x8 SRAM bitcell array tiled using Substrate.

The bitcell array has several structural features commonly found in SRAM bitcell arrays:

- One horizontal tap row for every 8 rows of cells.
- One vertical tap column for every 4 columns of cells. Note that this array was generated for a column mux ratio of 4. For a column mux ratio of 8, the tap columns would be placed every 8 cells.
- Special row end, column end, and corner cells.
- Each SRAM bitcell shares bitline and power contacts with its neighbors. As a result, each cell in a row is reflected horizontally with respect to the cell preceeding it. Similarly, each cell in a column is reflected vertically with respect to the cell above it.

The requirements for special edge and corner cells makes this a natural fit for a ninepatch tiler. To use the ninepatch tiling API, we must first partition the bitcell into the nine ninepatch regions (center, four edges, and four corners).

The center cell is the unit that tiles in both the horizontal and vertical dimensions. For the example shown above, the center cell contains a horizontal tap row, 8 rows of 4 bitcells, and a vertical tap column. Tiling this cell will produce an array with the desired frequency of taps (one horizontal tap every 8 rows, and one vertical tap every 4 columns). An image of the center cell is shown in 3.3. Note that we have placed taps at the top and left edges of the cell. We account for this when designing the edge and corner cells.

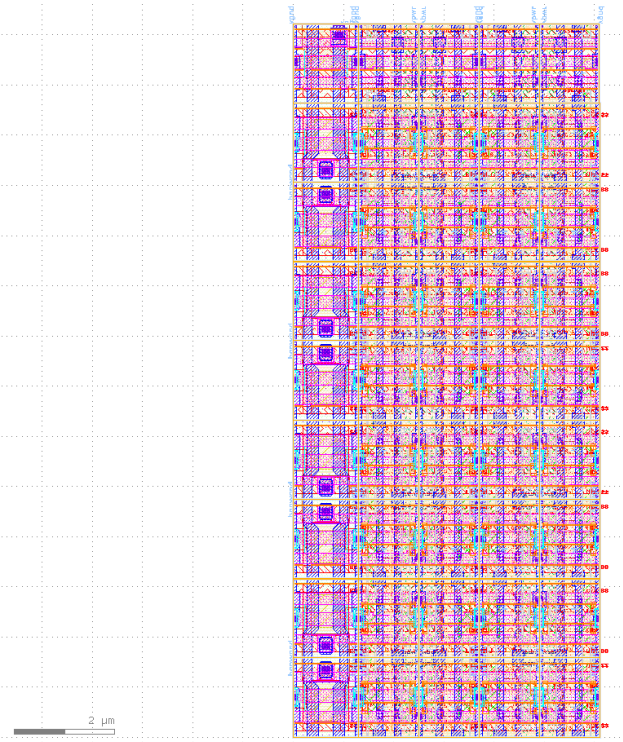
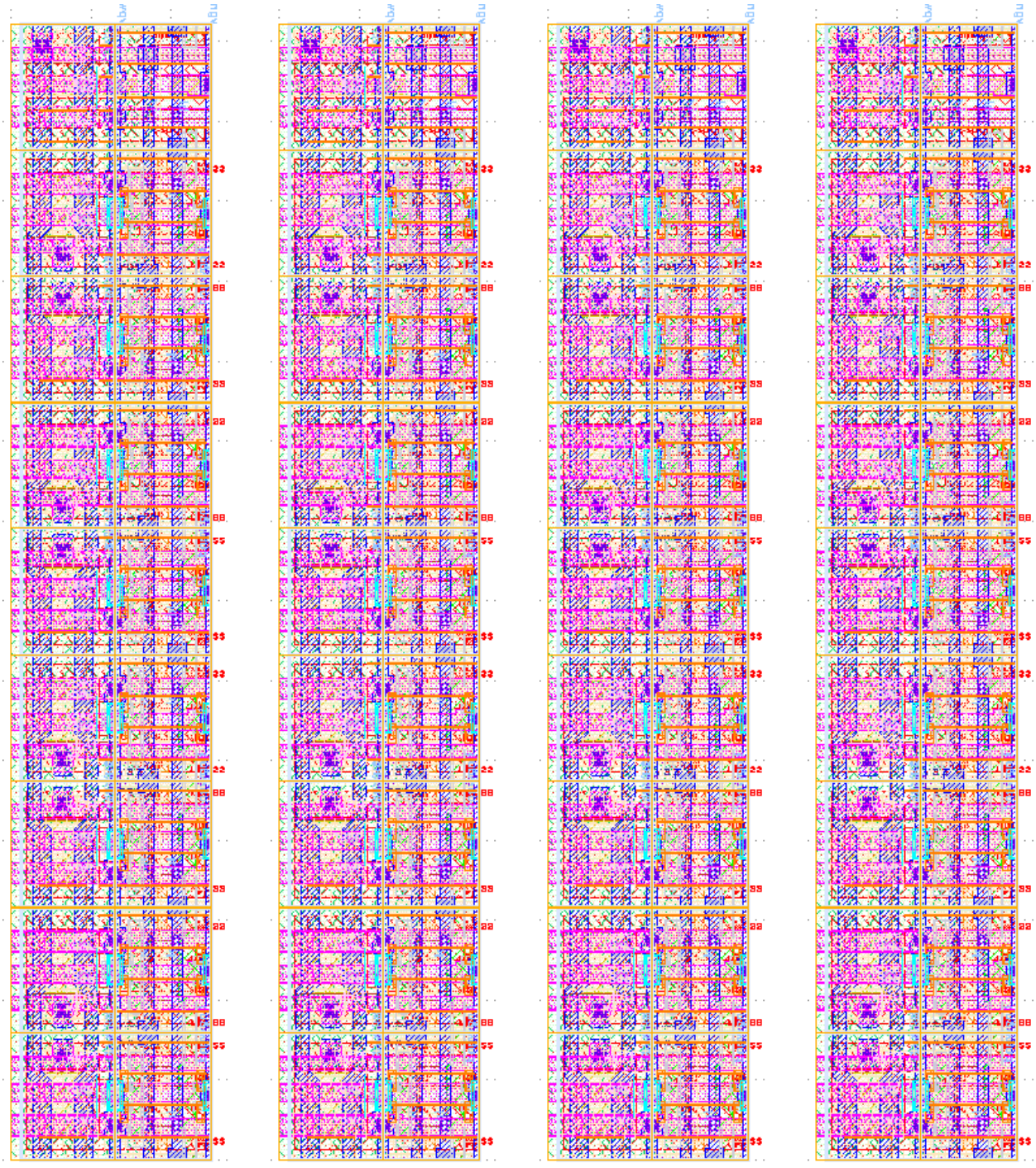


Figure 3.3: The center ninepatch cell for the bitcell array shown in 3.2.

The remainder of the ninepatch tiles are shown in 3.4. Since the center cell contains taps at the left and top edges, the left/top ninepatch cells do not contain internal tap structures, whereas the right/bottom cells do.



(a) Left edge (b) Left edge (c) Left edge (d) Left edge

Figure 3.4: The off-center ninepatch cells for the bitcell array shown in 3.2.

Once we have generated layouts of the ninepatch cells, tiling them using Substrate's `NpTiler` is fairly straightforward:

```
let tiler = NpTiler::builder()
    .set(Region::CornerUl, &corner_ul)
    .set(Region::Left, &left)
    .set(Region::CornerLl, &corner_ll)
    .set(Region::Top, &top)
    .set(Region::Center, &center)
    .set(Region::Bottom, &bot)
    .set(Region::CornerUr, &corner_ur)
    .set(Region::Right, &right)
    .set(Region::CornerLr, &corner_lr)
    .nx(nx)
    .ny(ny)
    .build();
```

Each of the nine ninepatch region tiles are composite cells; they each contain multiple foundry-provided primitive cells. To generate these tiles, we use Substrate's `GridTiler`.

For example, 3.5 shows a portion of the code for generating the left edge cell.

```
let cell_row: Vec<OptionTile> = into_vec![rowend_replica, cell];
let cell_opt1a_row: Vec<OptionTile> = into_vec![rowenda_replica, cell_opt1a];
let hstrap: Vec<OptionTile> = into_vec![rowend_hstrap, hstrap];

let mut grid = Grid::new(0, 0);
grid.push_row(hstrap);
for _ in 0..self.params.hstrap_ratio / 2 {
    grid.push_row(cell_opt1a_row.clone());
    grid.push_row(cell_row.clone());
}

let mut grid_tiler = GridTiler::new(grid);
```

Figure 3.5: A code snippet for generating the left edge cell of an SRAM bitcell array.

This code:

1. Creates three rows of 2 cells each: one normal bitcell row, one bitcell row with slight modifications to geometry for OPC, and one row for taps. These rows are just collections of cells; they are not yet placed in the actual cell layout.

2. Adds the tap row as the first (topmost) row in the grid. This row will align with the topmost row of the ninepatch center cell (3.3), which is also a tap row.

3. Adds alternating rows of regular bitcells and modified-OPC bitcells. Eight rows are added in total, since `self.params.hstrap_ratio` is 8 here. As is evident from the code, the frequency of horizontal tap rows can easily be adjusted.

4. Creates a `GridTiler` instance to place all of the cells.

The lowest level of cells in the hierarchy (eg. `cell`, `cell_opt1a` in the code above) are imported into Substrate using hard macros (2.3).

In addition to tiling geometry, we also expose relevant ports from each of the tilers. This allows other generators to figure out how to connect to the bitcell array. For example, we expose ports for each of the wordlines, bitlines, and power strap connections. The code to do this is omitted for brevity, but can be found in the SRAM22 source code.

Although bitcell tiling is not fundamentally a hard problem, we believe that the approach of decomposing the problem into reasonably intuitive parts (such as grid and ninepatch tiling) is better and more easily understood than the alternative, "simpler" method of writing a double for loop with edge cases to handle the array edge cells and periodic taps.

Precharge

The precharge cell illustrates how to use Substrate to produce compact, tileable layouts. The SRAM bitcell width in the Skywater 130nm process is 1.2 µm. The minimum metal 1 line and space is 0.28 µm, meaning that at most 4 M1 tracks will fit within the bitcell pitch. However, since minimum-sized vias are wider than the minimum M1 width, we are practically limited to around 3 M1 tracks. There are four signals involved in the precharge circuit: the bitline, complementary bitline, VDD, and active low precharge enable. Thus, routing performance is critical.

The final precharge cell is shown in 3.6.

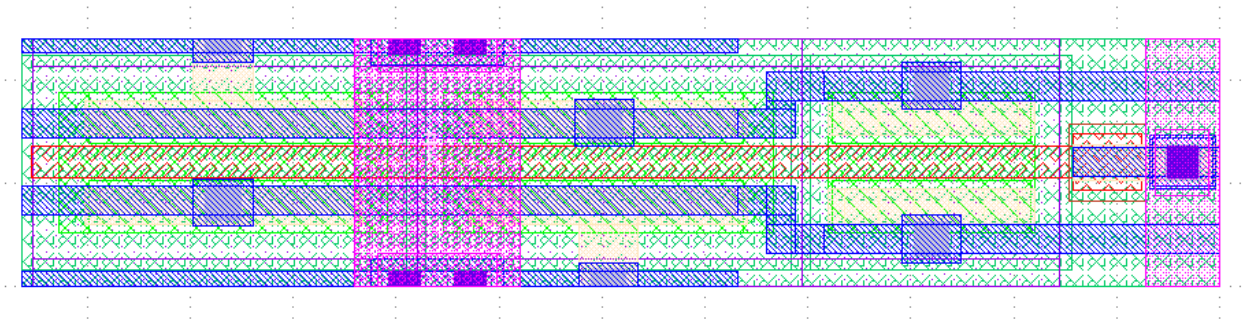


Figure 3.6: The pitch-matched precharge cell in SRAM22.

Before we describe how SRAM22 generates this layout, we first highlight some important features:

- VDD is routed into the cell from metal 2 (pink) and drops onto split tracks on metal 1 (blue). The split tracks of adjacent cells abut to form a full track.
- The bitlines jog slightly to make space to enable routing precharge enable to the gates of the precharge devices.
- The total capacitance (and therefore total wire length) on the bitline and bitline complement nets must be approximately equal.

To generate the precharge cell layout, we start by instantiating the three precharge transistors, as shown in 3.7.

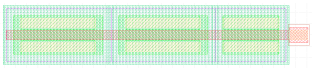


Figure 3.7: The three base transistors of a precharge cell.

```
let params = LayoutMosParams {
  skip_sd_metal: vec![vec![]; 3],
  deep_nwell: true,
  contact_strategy: GateContactStrategy::SingleSide,
  devices: vec![
      MosParams {
          w: self.params.equalizer_width,
          l: self.params.length,
          m: 1,
          nf: 1,
          id: mos.id(),
      },
      MosParams {
          w: self.params.pull_up_width,
          l: self.params.length,
          m: 1,
          nf: 1,
          id: mos.id(),
      },
      MosParams {
          w: self.params.pull_up_width,
          l: self.params.length,
          m: 1,
          nf: 1,
          id: mos.id(),
      },
  ],
};

let mut mos = ctx.instantiate::<LayoutMos>(&params)?;
mos.set_orientation(Named::R90);
```

Figure 3.8: A snippet of code for generating the precharge transistors shown in 3.7.

Next, we allocate M1 routing tracks. At the left edge of the cell, the outermost half-tracks are used for routing VDD; the inner two tracks are the bitlines. Towards the right side of the cell, the bitline tracks move to make space for the gate contact.

These two routing regions are encoded using a `FixedTracks` object in Substrate, connected by a `SimpleJog`. The `FixedTracks` object produces tracks with a fixed line and space, centered in a given routing region. It also accepts boundary conditions, which permit

the user to place full tracks, half tracks, full spaces, or half spaces at the edge of the routing region.

The `SimpleJog` object allows the user to specify an arbitrary number of source tracks, and an equal number of destination tracks. It then jogs each source track to the corresponding destination track.

A code snippet (lightly edited for clarity) and image of this stage of precharge cell generation are shown in 3.16 and 3.9.

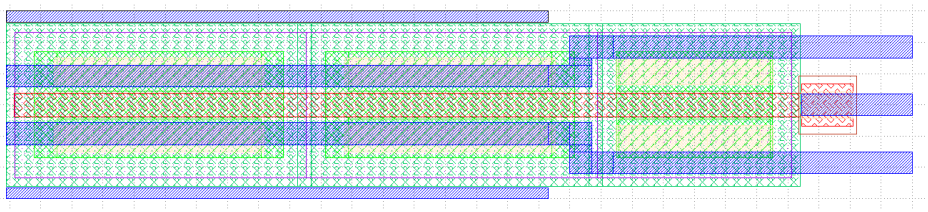


Figure 3.9: Precharge cell metal 1 routing.

```
let in_tracks = FixedTracks::from_centered_tracks(CenteredTrackParams {
    line: 140,
    space: 230,
    span: Span::new(0, 1_200),
    num: 4,
    lower_boundary: Boundary::HalfTrack,
    upper_boundary: Boundary::HalfTrack,
    grid: 5,
});
let out_tracks = FixedTracks::from_centered_tracks(CenteredTrackParams {
    line: 140,
    space: 230,
    span: Span::new(0, 1_200),
    num: 3,
    lower_boundary: Boundary::HalfSpace,
    upper_boundary: Boundary::HalfSpace,
    grid: 5,
});

// ...

let mut orects = Vec::with_capacity(dsn.out_tracks.len());
for i in 0..dsn.out_tracks.len() {
    let top = if i == 1 { gate.top() } else { cut };
    let rect = Rect::from_spans(dsn.out_tracks.index(i), Span::new(0, top));
    ctx.draw_rect(dsn.v_metal, rect);
}

let jog = SimpleJog::builder()
    .dir(Dir::Vert)
    .src_pos(cut)
    .src([dsn.out_tracks.index(0), dsn.out_tracks.index(2)])
    .dst([dsn.in_tracks.index(1), dsn.in_tracks.index(2)])
    .line(dsn.v_line)
    .space(dsn.v_space)
    .layer(dsn.v_metal)
    .build()
    .unwrap();

for i in 0..dsn.in_tracks.len() {
    let rect = Rect::from_spans(
        dsn.in_tracks.index(i),
        Span::new(jog.dst_pos(), bbox.height()),
    );
    ctx.draw_rect(dsn.v_metal, rect);
}
```

Next, we draw vias connecting the metal 1 tracks to transistor sources/drains/gates. We also draw a power strap on metal 2 and drop vias to the VDD tracks on metal 1. Sample code for drawing a via is shown in 3.11. The `ViaExpansion::LongerDirection` flag shown in that figure tells the via generator that the via need not be contained in the overlap of the geometry on the top and bottom layers; it can expand along the longer direction of the overlap region. This produces the 2x1 via arrays seen in 3.12. If no expansion is permitted, Substrate fills the overlap region with as many vias as possible. At least one via will be drawn, even if it does not fit entirely within the overlap region.

The precharge cell after drawing all vias and the power strap is shown in 3.12.

```
let mut via1 = ViaParams::builder()
  .layers(dsn.v_metal, dsn.h_metal)
  .expand(ViaExpansion::LongerDirection)
  .geometry(rects[0].double(Side::Left), stripe)
  .build();
let via = ctx.instantiate::<Via>(&via1)?;
ctx.draw(via)?;
```

Figure 3.11: Sample code snippet for drawing a via.

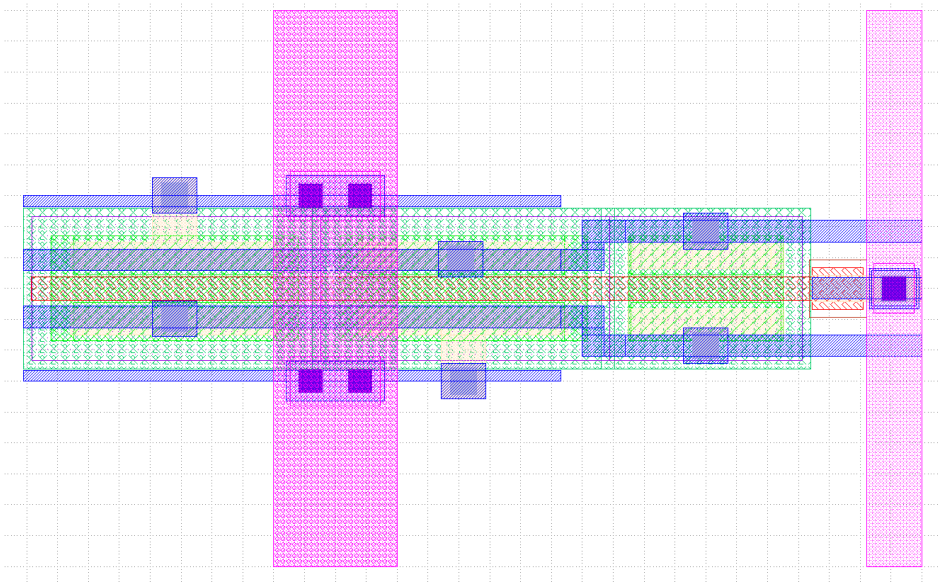


Figure 3.12: The precharge cell after drawing vias and a power strap.

Finally, as shown in 3.13, we trim the precharge cell so that it can be tiled (using the tiling APIs described previously).

```
let bounds = ctx.brect().with_hspan(Span::new(0, dsn.width));
ctx.trim(&bounds);
```

Figure 3.13: Trimming the precharge cell.

There is an alternate convention for tiling cells, which is to skip the step of trimming the cell, and instead allow adjacent cells to overlap. However, the convention of trimming cells is more convenient here for two reasons:

1. Substrate tilers place cells according to their bounding boxes. Tiling untrimmed cells would require wrapping them in a `LayerBbox` tile (see 2.3) or manually specifying the bounding box to the tiler. Trimming the cell is more convenient than either of these options.

2. Special end cells are required to provide taps. So even if we were to tile untrimmed cells, we would still need to specify custom cells for the array edges and tap columns.

The precharge cells illustrate how Substrate APIs can be used to produce very compact layout. Much of the SRAM peripheral circuitry is generated in a similar manner. An image of the peripheral circuitry layout is shown in 3.14. We omit specific discussion of the generators for the rest of the column components, since they are similar in principle to the precharge cell.

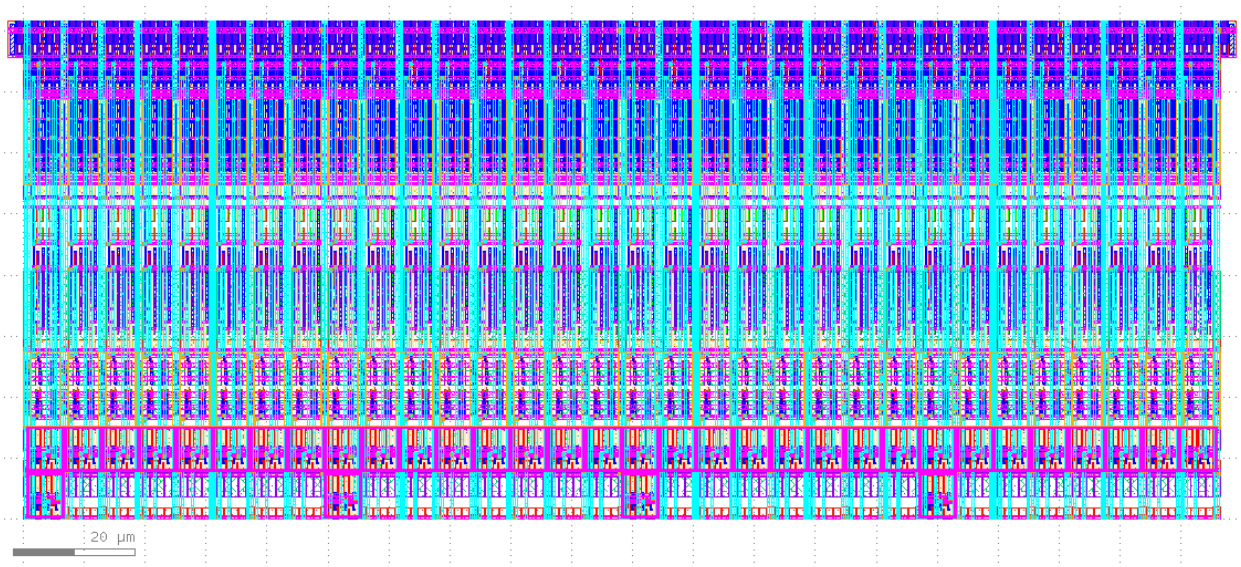


Figure 3.14: Tiled column peripheral circuitry. The precharge cells occupy the topmost row.

Control Logic

The control logic is a small (≈ 100) set of logic gates that control the sequence of events during read and write operations.

Despite being entirely digital, using a traditional digital flow to lay out the control logic is undesirable:

- The control logic is pulsed, unlike the typical digital circuits for which these tools are optimized.
- Minimizing area is important.
- SRAM22 tries not to depend on commercial tools, and open source digital tool flows are not very mature.
- Dependency on external tools would make SRAM22 more difficult and less convenient to use.

Therefore, we opt instead to place and route the control logic entirely within SRAM22 and Substrate.

The flow for using digital standard cells in Substrate is:

1. Import the desired standard cell libraries into Substrate.
2. Place the standard cells, often using a `GridTiler` (2.3) with tiles taking their bounding box from a "PR boundary" layer.
3. Select routing layers and define a routing grid for each layer.
4. Route between the placed standard cells.

The fourth step is often done with a combination of manual routing and automatic routing. SRAM22 uses manual routing when drawing routes than run solely on the local interconnect layer in SKY130. For longer distance routes that span other metal layers, SRAM22 defers routing to the automatic router in Substrate.

We will focus on the use of the automatic router in this section, since this feature is, as far as we are aware, not present in other analog generator frameworks. The router implementation in Substrate is called `GreedyRouter`, as it creates routes as they are specified, rather than trying to find a globally optimal routing solution.

To initialize the router, we specify a routing region and a set of routing layers. Each routing layer holds the line and space of tracks on that layer, a preferred routing direction, and a reference to a PDK layer. Router intialization is shown in **??**.

```
let mut router = GreedyRouter::with_config(GreedyRouterConfig {
    area: group.brect().expand(1840),
    layers: vec![
        LayerConfig {
            line: 320,
            space: 140,
            dir: Dir::Horiz,
            layer: m1,
        },
        LayerConfig {
            line: 320,
            space: 140,
            dir: Dir::Vert,
            layer: m2,
        },
    ],
});
```

Figure 3.15: Greedy router initialization with two metal layers (M1 and M2).

The current implementation of `GreedyRouter` only permits on-grid routing. However, cells being routed may have off-grid pins. These pins must be jogged onto the routing grid before asking the router to route from/to them. Substrate provides an `expand_to_grid` function for doing this. An example usage is shown in **??**. Users specify a rectangle representing the off-grid pin and an expansion strategy. The router will automatically find an appropriate on-grid point to which the off-grid pin can be connected, consistent with the expansion strategy. Possible expansion strategies include:

- Minimum area: connect to the routing grid using the minimum amount of extra metal area.
- Top/bottom/left/right: connect to the nearest grid point to the given side of the source pin.
- Upper left/lower left/upper right/lower right: connect to the nearest grid point off of the given corner of the source pin.

```
let clkp_out = router.expand_to_grid(
    clkp_out_via.layer_bbox(m1).into_rect(),
    ExpandToGridStrategy::Minimum,
);
router.occupy(m1, clkp_out, "clkp")?;
```

Figure 3.16: Expanding the (pulse generator output) pin to the routing grid.

Substrate also has more sophisticated routines for bringing off-grid geometry onto the routing grid. For example, users can ask the Substrate router to jog a bus of off-grid pins onto adjacent routing tracks. The router will calculate the minimum necessary amount of spacing in the routing layer's preferred routing direction. The router logic can handle a wide range of off-grid pitches, and can handle cases where the off-grid bus is not centered with the target on-grid tracks. An example of the geometry generated by this procedure is shown in 3.17.

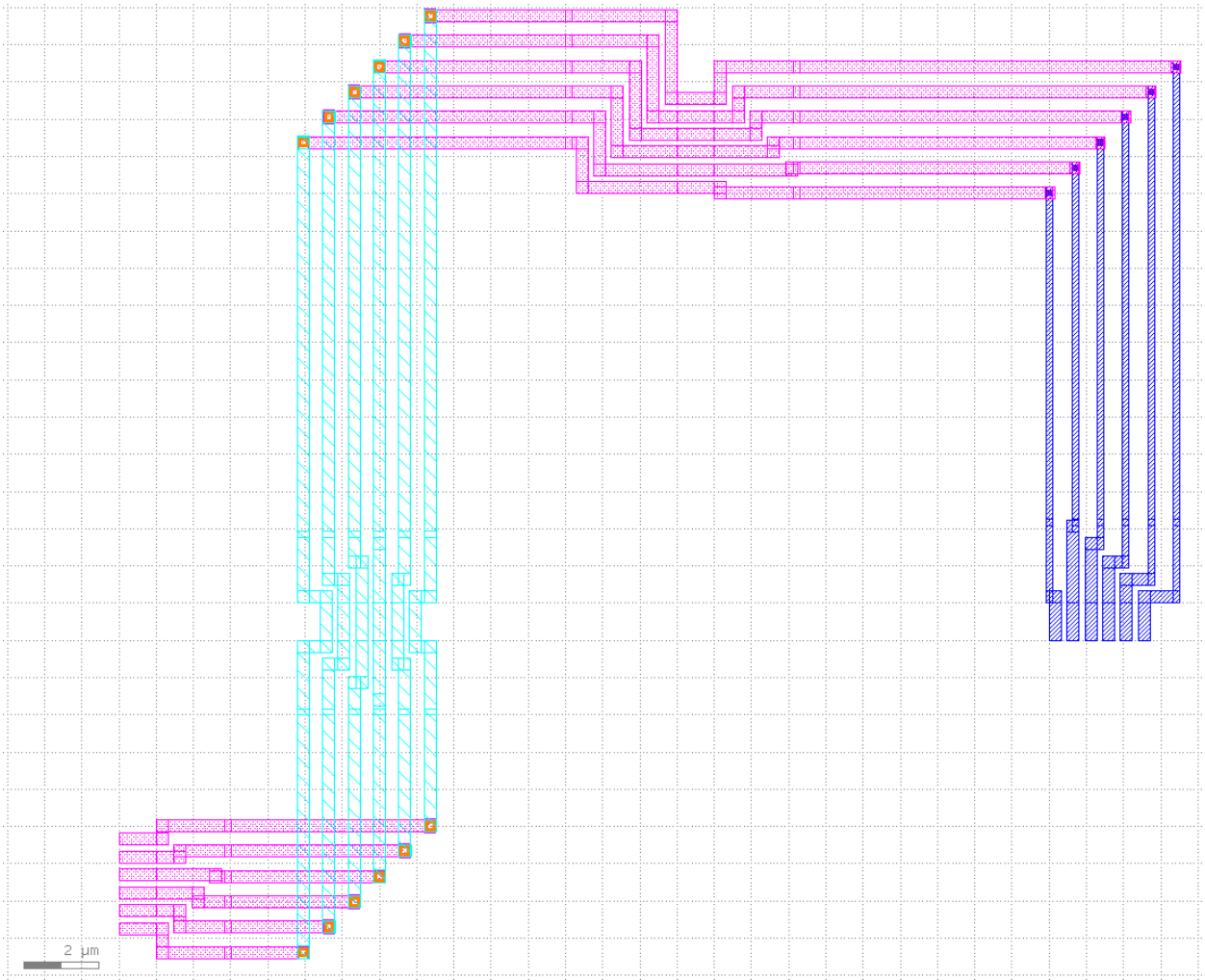


Figure 3.17: A sample of the geometry automatically generated by the router to bring off-grid pins to the routing grid.

Bringing off-grid pins to the routing grid is by far the most tedious part of generating layouts based on digital standard cells. Further improvements to Substrate have the potential to make this process easier.

Once all geometry is on-grid, routing becomes very easy (see 3.18).

```
router.route_with_net(ctx, m1, clkp_out, m1, clkp_in_1, "clkp")?;
router.route_with_net(ctx, m1, clkp_out, m1, clkp_in_2, "clkp")?;
router.route_with_net(ctx, m1, clkp_out, m1, clkp_in_3, "clkp")?;
```

Figure 3.18: Routing the clock pulse generator output to three consumers.

By issuing several of these routing commands, the full layout can be programatically routed. Extracting these routing paths from a schematic view is possible in principle, but has not been implemented at the time of writing.

The fully-routed layout of SRAM22's control logic is shown in 3.19. The output pins are placed towards the upper right for convenience in top-level routing.

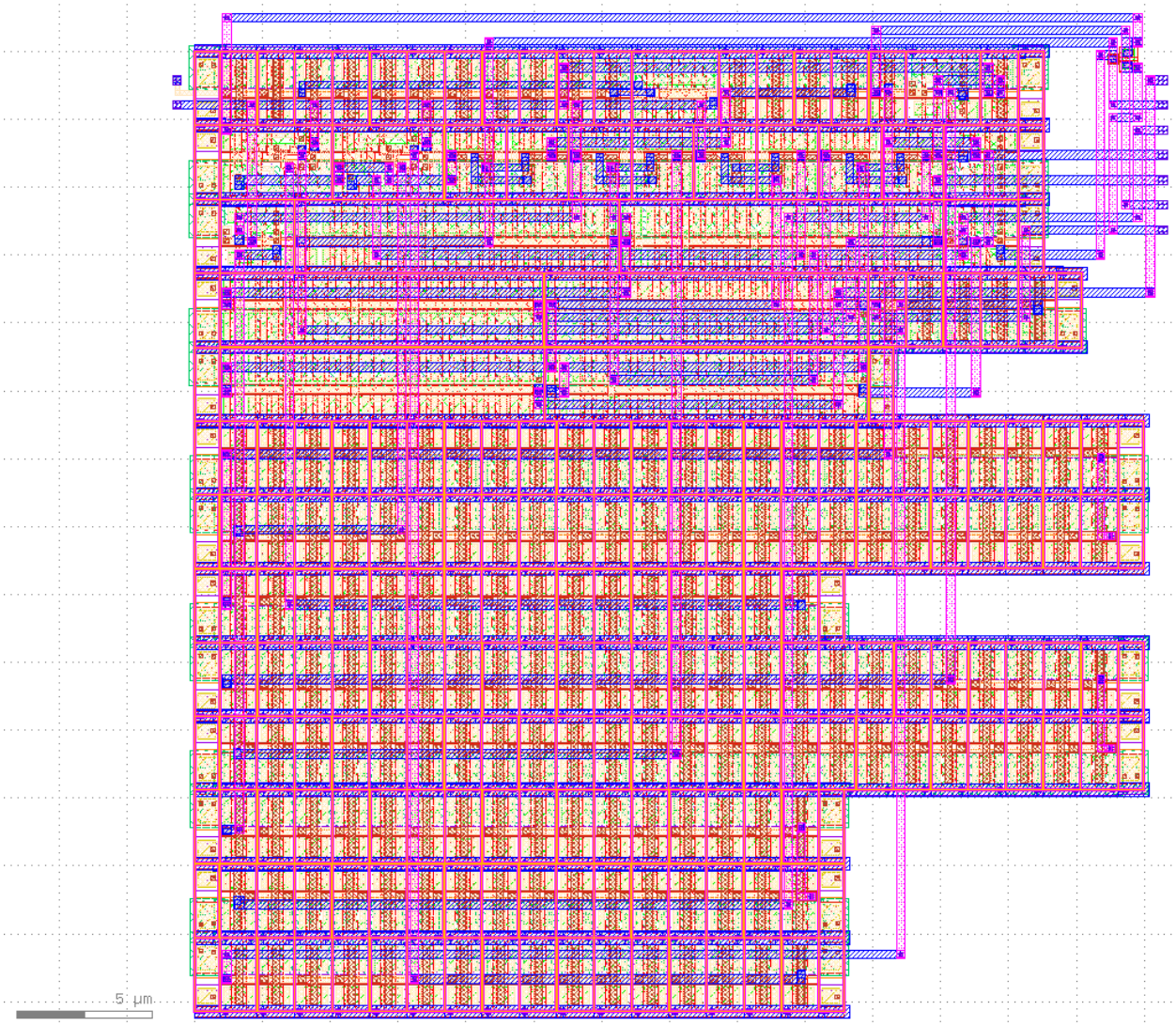


Figure 3.19: The control logic cell in SRAM22. Most routing is done via the automatic router in Substrate.

3.3 Conclusion

Additional Features

Substrate has many features beyond the ones described in this thesis. In particular, Substrate provides APIs for functional verification and for verifying digital timing constraints (eg. setup/hold times) in contexts where analog workflows are more convenient. The latter feature was used to design and verify a schematic-level tree serializer generator intended for die-to-die links.

Future Work

There are a wide variety of directions for future work. We suggest a few here.

1. Designing an intermediate representation for representing circuits as graph models.
2. Smarter algorithms for analog routing. In particular, it would be nice to have an automatic router capable of generating symmetric/matched differential routing, while understanding which nets are aggressors and which are victims.
3. Easing the process of integrating analog IP into digital flows, a process that currently requires setting up flows for several tools not often used in a purely-analog workflow.

Bibliography

[1] B.S. Amrutur and M.A. Horowitz. "A replica technique for wordline and sense control in low-power SRAM's". In: *IEEE Journal of Solid-State Circuits* 33.8 (1998), pp. 1208–1219. DOI: 10.1109/4.705359.

[2] 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)*. Nov. 2013, pp. 74–81. DOI: 10.1109/ICCAD.2013.6691100.

[3] Tonmoy Dhar et al. "The ALIGN Open-Source Analog Layout Generator: v1.0 and Beyond (Invited talk)". In: *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. Nov. 2020, pp. 1–2.

[4] Mark Horowitz. "Why design must change: Rethinking digital design". In: *2010 Design, Automation and Test in Europe Conference and Exhibition (DATE 2010)*. Mar. 2010, pp. 791–791. DOI: 10.1109/DATE.2010.5457119.

[5] Benjamin Prautsch et al. "Template-Driven Analog Layout Generators for Improved Technology Independence". In: *ANALOG 2018; 16th GMM/ITG-Symposium*. Sept. 2018, pp. 1–6.

[6] Zhongkai Wang and Elad Alon. "Analog Generators for SerDes Clock Generation and Distribution". PhD thesis. EECS Department, University of California, Berkeley, May 2023. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-36.html.