

**Sponge Paint – A Procedural Layout Library in
Java**

by

Gong Ke Shen

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2000

© Massachusetts Institute of Technology 2000. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 5, 2000

Certified by.....
Krste Asanovic
Assistant Professor
Thesis Supervisor

Accepted by.....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Sponge Paint – A Procedural Layout Library in Java

by

Gong Ke Shen

Submitted to the Department of Electrical Engineering and Computer Science
on May 5, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We have developed **Sponge Paint**, a Java library of tools that allow the user to layout microchip designs by writing Java programs. We chose to use a general purpose language because it is non-proprietary and makes it easy to integrate Sponge Paint into other applications. Sponge Paint designs can also be made independent of process technology. As an illustration, we built a datapath generator using Sponge Paint. Current problems in datapath generators include dealing with irregularity in a bit-slice structure and bus routing. The Sponge Paint datapath generator deals with these problems through abstraction and simple heuristics, and by allowing the designer to add arbitrary Java code to handle exceptional cases. The generator has a three tiered structure: basic leaf cells, component builders and the grid. Basic single-bit leaf cells are provided as hand-crafted layout. Component builders specialize in constructing multi-bit structures such as muxes and latches. These builders can be arbitrary hierarchical and recursive Java code. The grid is a central 2-D matrix of virtual cell positions. It coordinates component builders to prevent wiring conflicts between builders. The central grid has variable length rows and variable width columns which accommodate irregularities in the datapath. The grid is responsible for global elements such as buses, and abstracts away exact coordinates of leaf cells. Track assignment for buses is performed using a simple greedy algorithm to minimize total height. The datapath generator has a final phase where the grid provides builders with absolute coordinates for module layout and bus track positions for final placement and routing.

Thesis Supervisor: Krste Asanovic
Title: Assistant Professor

Acknowledgments

Well, I have to say this is the most fun part of the writing, because I can write just like I talk and not get in trouble for that. I would like to first clarify that the following gratitudes do not follow any type of order.

I would like to thank my parents for keeping me on the road to academic providence, mainly by convincing me that I wasn't happy because I didn't study enough. My sweetheart Chris for infinite patience and for believing in me.

I would also like to thank my schooling fishes for not dying during the time of my thesis writing, since I would have spent hours researching for new fish instead of researching for my thesis. I know my writing sucks, which is why I very appreciate Andrew G., Andrew M. and Ronny K. for their help in fixing my writing. My boots would like to thank Dave for being so supportive. Seriously, Dave's been a great friend.

Mad props out to Krste for being such a kick-ass advisor! He is truly one of the most understanding and knowledgeable advisors out there.

Thank you all!

Contents

1	Introduction	9
2	Sponge Paint Basics	12
2.1	Design Goals and Overview	12
2.2	Primitive Components	14
2.2.1	Rectangle	14
2.2.2	Via	15
2.2.3	Label	15
2.2.4	Use	16
2.3	Complex Components	16
2.3.1	Cell	17
2.3.2	Terminal	17
2.4	Abstraction Tools	17
2.4.1	Input Cell Descriptions	17
2.4.2	Output Layout	18
2.4.3	Design Rules	19
3	Sponge Paint Datapath Generator	21
3.1	How Datapaths Fit into VLSI Design	21
3.2	Design Problems	23
3.3	Previous work	23
3.4	Datapath Layout Generator Overview	26
3.5	The Grid	27

3.5.1	Grid Border	29
3.6	Builders	30
3.6.1	How To Build a Builder	32
3.6.2	A special builder: Bus Ripper	34
3.7	Buses	35
3.7.1	Bus layout	36
3.7.2	Conflicting wires	37
3.8	The <i>done</i> process	38
4	Examples of using Sponge Paint	42
4.1	Datapath Generator	43
4.1.1	Implementation	44
4.2	Other examples	45
4.2.1	Vertical Bus Ripper	45
4.2.2	32-bit Shifter	46
5	Conclusion and Future Improvements	50
5.1	Future Improvements	51
A	Programs	53
A.1	H-tree Java Code	53

List of Figures

2-1	Interaction Between Sponge Paint Components	13
2-2	Class inheritance diagram of Sponge Paint primitives.	14
2-3	The Sponge Paint output for a 4 level H-tree.	15
2-4	Some examples of Sponge Paint primitives.	16
2-5	A sample Magic input file for a 2-input mux.	19
3-1	Representation of a datapath.	22
3-2	The layout of a 1-bit 2-input mux.	27
3-3	A 1-bit 2-input mux.	28
3-4	A 4-bit 2-input mux.	29
3-5	Control flow of a layout design using Sponge Paint.	30
3-6	A Sample Grid and its Contents	31
3-7	Class Dependence Diagram for Bus Related Classes	35
3-8	Step 1. An Empty Grid	39
3-9	Step 2. Add components into the Empty Grid	39
3-10	Step 3. Add virtual connections between symbolic components.	40
3-11	Step 4. Crude bus layout after user calls <i>done()</i>	40
3-12	Step 5. Grid scaling rows and columns	41
3-13	Step 6. Exact bus layout done by the Grid and builders connect terminals to the buses	41
3-14	Final Step. Include manual non-repetitive components such as select lines and their labels.	41
4-1	Sponge Paint package hierarchy	42

4-2	A CPU bypass schematic designed by the user.	43
4-3	A CPU bypass generated by the Sponge Paint datapath generator. . .	47
4-4	The Sample Output of a Bus Ripper Builder.	48
4-5	A sample 32-bit shifter generated by Sponge Paint.	49

List of Tables

Chapter 1

Introduction

Computer Aided Design (CAD) tools continue to grow in complexity and function as modern computer chip production demands require an increasingly fast design process of macro-cells. Expert designers usually design the macro-cells manually using current CAD tools and this process dominates the design time [7].

Most current CAD tools take graphical input from the user. Manual layout with a graphical user interface is easy to see and use. Some graphics tools can extract a schematic from a layout, which can be very helpful in analyzing a design [1]. However, graphical manual input becomes cumbersome, time-consuming and tedious for human designers especially for designs involving repeated components or when small design constraints change.

The complete automation of layout is known as automatic synthesis place and route. Of course, complete automation saves the designer a tremendous amount of work. There are a few disadvantages to complete automation though. For example, it takes a very long time to run a place and route algorithm, and the resulting layouts are never as efficient as manual layouts. With complete automation, users lack control over where components end up.

Between complete automation and manual layout, there is procedural layout. The user decides the general structure of the layout as he does with manual layouts, but the tedious details are eliminated. Thus, procedural layout incorporates the advantages of the other two methods. Procedural layout is most applicable for regular

circuit designs such as datapath and memory arrays. Irregular structures such as control logic already have efficient generators on the market, and are best handled by fully automatic systems such as place and route. Since Sponge Paint has a general input/output abstraction, layouts generated by other programs can be incorporated into a final Sponge Paint layout.

Sponge Paint is a Java library of tools that allows designers to describe procedural layouts in Java. Sponge Paint was designed to give the user maximum design flexibility while allowing easy insertion of new functions. Sponge Paint puts power in users' hands, making it easy for them to insert into the library almost any new function important for their particular application.

Sponge Paint is technology independent. It achieves this independence by using pseudo-symbolic layouts. This independence further aids users by abstracting away tedious details of different technology constraints. The eventual goal of Sponge Paint is to simplify the lives of digital chip designers by allowing the generation of an entire chip layout with just one program.

The majority of existing procedural layout programs use proprietary languages, such as Cadence's SKILL, to script the layouts. SKILL is a LISP-like language that is arguably the most widely used scripting language for hardware layouts. It allows the users to "customize and extend your design environment" while automatically handling many traditional system programming operations such as memory management [9]. However, SKILL programs are only executable in the Cadence environment, thus decreasing extensibility.

We decided to build Sponge Paint using Java, a general purpose language that also handles exceptions and has an extensive User Interface library such as Java Swing. Furthermore, Java is portable and freely available. Tools for tracing, debugging, documenting, and profiling are already available in numerous debugging suites as well as operating systems. The flexibility that comes from using Java serves to broaden Sponge Paint availability.

The rest of this thesis is structured as follows. Section 2 describes the Sponge Paint basics including primitive types and their usage. Section 3 describes the design

of the Sponge Paint datapath generator. As an example for using Sponge Paint, we will build a CPU bypass network. Section 4 outlines some other examples of using Sponge Paint. We conclude in section 5 with future improvements.

Chapter 2

Sponge Paint Basics

2.1 Design Goals and Overview

In real life, painting with a sponge allows the creation of quite intricate patterns with very simple strokes. The goal of Sponge Paint as a procedural layout library is to achieve the same simplicity and efficiency with VLSI chip design, i.e. to be able to lay out complicated chips with minimum effort from the human designer.

The main design goals for Sponge Paint include:

1. **Flexibility.** Sponge Paint must be easily extensible and scalable so that users can add to the library's functionalities. This is very important as it allows designers to deal with the fast-paced VLSI chip design industry by modifying Sponge Paint accordingly.
2. **Generality.** We wanted to avoid proprietary languages and technology-specific assumptions because they may tend to restrict usage of Sponge Paint. By using a general purpose programming language, Sponge Paint maximizes portability, and makes it easier to add features later. By eliminating assumptions about the underlying technology, Sponge Paint is easily convertible when new technologies become available.
3. **Incremental and Hierarchical structure.** VLSI layouts are known to be large and time-consuming to generate. Generating layouts in an incremental

and hierarchical fashion saves time and changes a linearly increasing production time scale into a logarithmic time scale as the layout size multiplies.

With these goals in mind, Sponge Paint adopts a three-tiered object representation. Each layer serves a different purpose, and their interactions define the data control flow. The top layer is the designer who knows exactly what components to use and the order in which to use them. However, he neither cares about the exact locations of these components nor about obtaining the absolute optimum wiring between connecting terminals. The designer writes a top level Java program that would lay down components.

The designer also creates the leaf cells used. Leaf cells are the basic building blocks of most layouts, and is counted as the bottom layer. The middle layer consists of the Sponge Paint primitives. They provide an easy way for users to describe components and locations. An interesting circularity exists between the top layer and the bottom layer – the output of the top layer user program can be used as input leaf cells. Figure 2-1 shows the interactions between these layers.

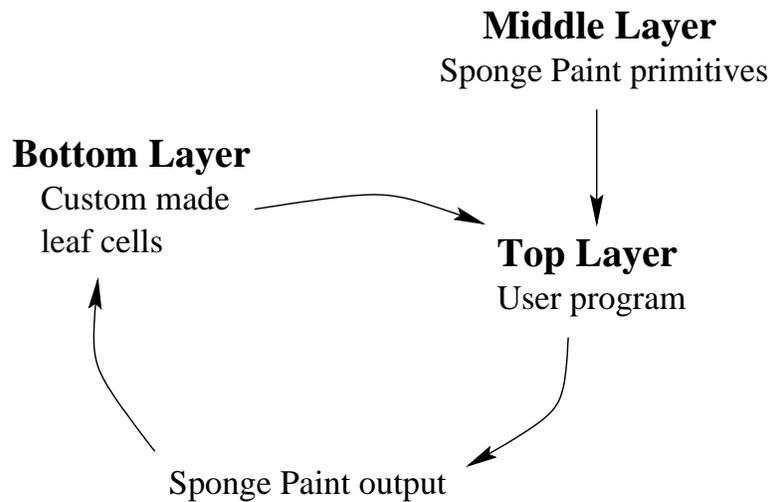


Figure 2-1: Interaction Between Sponge Paint Components

This chapter describes the middle layer primitives and shows examples of their usage.

2.2 Primitive Components

The `SP_Base` class represents items that can be directly translated to a real coordinated entity. These base objects are the building blocks of more complex structures. See Figure 2-2 for the class inheritance diagram.

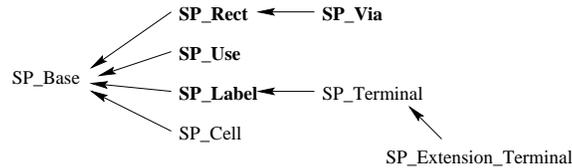


Figure 2-2: Class inheritance diagram of Sponge Paint primitives.

2.2.1 Rectangle

The most basic component in Sponge Paint is the `SP_Rect` object that represents rectangles. A Sponge Paint rectangle is a box defined by its lower left and top right corners. Rectangles have a layer property representing the material of the rectangle. Possible materials include *metal1*, *metal2*, a via connection, etc. Any layout can be done using rectangles only, making it the building block of everything.

Sponge Paint provides an `SP_Pen` class as an easy interface for the user to draw rectangles, i.e. a pen will draw in any of the four directions for a given number of units. Pen locations are described by relative coordinates. For example, to draw to the right for 5 units from the pen's current location, the user would call the method `drawRight(5)` in the `SP_Pen` class (See Appendix A.1).

Using just the pen and the rectangle classes, we can draw a very useful h-tree with a simple “while” loop (See Figure 2-3). It has a special property that each leaf point is equidistant from the center of the fractal. The fanning nature of the fractal makes h-trees very useful for clock synchronization when it is laid on top of a chip, since every component on the chip will be close to a leaf point and will receive a clock signal at the same time if it was sent out from the middle of the fractal.

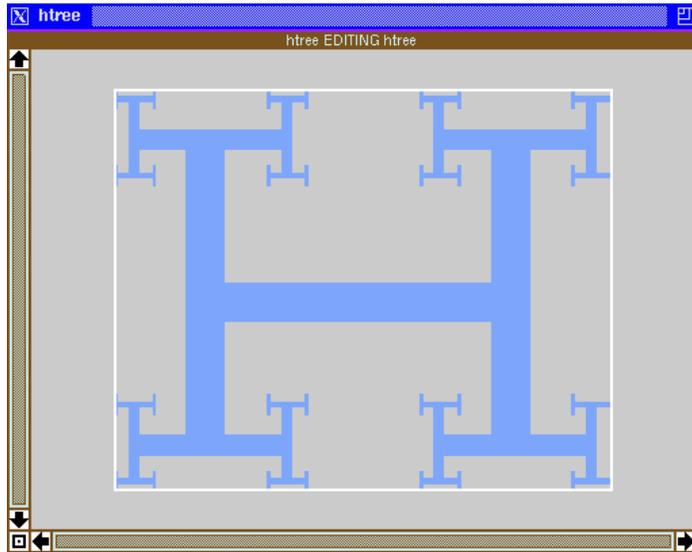


Figure 2-3: The Sponge Paint output for a 4 level H-tree.

2.2.2 Via

`SP_Via` objects represent vias and are special rectangles whose layer property represents metal contacts. Metal contacts are vertical junctions connecting different metal layers. For example, we would use a via to connect *metal1* with *metal2*, or *metal2* with *metal3*, etc. (See Figure 2-4).

2.2.3 Label

To make manageable layouts, the user will need a way to refer to connections and keep track of components in general. `SP_Label` is a special `SP_Base` object that represents text labels. It has a text field, a *layer*, and a *position* orientation. The text field is the name of the label. Normally, *layer* would have a value equivalent to “not applicable”, but if the label was for all wires on level 2, i.e. all *metal2* wires, *layer* would have the value of metal 2. The *position* is the orientation of the label text, i.e. whether or not

it appears vertical, horizontal, mirror-imaged, etc. when viewed with a layout editor such as Magic (See Figure 2-4).

2.2.4 Use

SP_Use objects allow the user to abstract away rectangles and use black box inclusion of other components. Given a leaf cell file from the user, an SP_Use object parses out information on the dimension of the box inclusion, label positions and texts (See Figure 2-4). It is a standard feature in layout managers to allow hierarchical inclusion of other components without flattening them out. Such inclusions prevent the size of the design layout from growing exponentially with the complexity of the components, and allow the user to build hierarchical structures easily.



Figure 2-4: Some examples of Sponge Paint primitives.

2.3 Complex Components

Complex components help organization and operation between Sponge Paint processes. Unlike primitives, complex components do not directly translate into real coordinates. They often contain and operate on primitives, or make it easier for

other classes to operate on primitives.

2.3.1 Cell

`SP_Cell` is a general container for all `SP_Base` objects that can be translated, including other `SP_Cell` objects. `SP_Use` and `SP_Label` can all be part of the contents of an `SP_Cell` object. This hierarchy is very natural especially in representing black box components for hierarchical layouts. It is also very convenient for grouping components and passing them between operations.

2.3.2 Terminal

Labels in base cell files are interpreted as locations for connections. As a result, they are represented as terminals in Sponge Paint. The `SP_Terminal` class inherits from the `SP_Label` class. Besides the label text, a terminal object also has to keep track of any assigned target connection. A target connection can be another terminal or a bus.

2.4 Abstraction Tools

The abstraction from technology has three divisions: input cell descriptions, output layout format and design rules. These divisions are completely independent of each other. The user can parse the input in a particular format and output in another. Having the input and output separate from the object representation of a layout is very flexible. If the designer chooses a different layout editor, all that is necessary to compensate is a matching output class. The object representation does not need to change at all to switch between different layout editors.

2.4.1 Input Cell Descriptions

Leaf cell files given by the user are translated into Sponge Paint objects. These files are processed by a parser that is knowledgeable about the formatting of the files,

which are dependent on the layout program that generated the leaf cell files. In most cases, the parser calculates the boundary of a leaf cell and creates an `SP_Use` object to represent it. If the technology is new to Sponge Paint, the user must provide an appropriate parser.

Currently, Sponge Paint only supports input files from Magic, a graphical layout program. There are several advantages to using Magic:

1. Magic files can be converted to GDS II format, which is the industry standard for design files.
2. Magic files are formatted, human readable and can be easily generated.
3. Magic has a built in design rule-checker for making sure that the design is valid for the given technology, making it very easy to verify the correctness of a layout.

Figure 2-5 shows a sample Magic input file. The Sponge Paint input abstraction layer would take such an input and pass it on to the parser appropriate to the current process. The parser will then parse the input file into an `SP_Use` object and `SP_Terminal` objects. The use object represents an *instance* of the input file, containing the dimensions of the object and the object filename. The dimensions are parsed from the *rect* lines by taking the minimum of the lower coordinates and the maximum of the upper coordinates.

Sponge Paint assumes that the label for a terminal resides at the same position as the terminal. Thus, the *rlabel* lines in the input file are translated into terminal objects. Sponge Paint relies on these terminal objects for starting and ending points of connections.

2.4.2 Output Layout

The output abstraction must obey the file formats of each technology supported by Sponge Paint. Sponge Paint currently outputs text files that can be read directly by Magic. Other display programs can be supported by inserting corresponding output

```

magic
tech scmos
timestamp 948396946
<< ntransistor >>
rect -61 -36 -58 -34
rect -61 -44 -58 -42
rect -61 -52 -58 -50
rect -23 -52 -20 -50
rect -61 -60 -58 -58
<< ptransistor >>
rect -46 -36 -43 -34
●
●
●
<< metall >>
rect -76 -37 -69 -33
rect -58 -41 46 -37
rect -58 -49 -46 -45

<< labels >>
rlabel metal1 -71 -35 -71 -35 3 in1
rlabel metal1 -71 -59 -71 -59 3 in2
rlabel metal2 -74 -31 -74 -31 4 cin1
●
●
●
rlabel metal2 -18 -58 -18 -58 1 GND2
rlabel metal2 -10 -58 -10 -58 1 VDD2
<< end >>

```

Figure 2-5: A sample Magic input file for a 2-input mux.

classes. We use Magic to translate the Sponge Paint output text file to an actual layout. This layout can then be converted to GDS II format which is the industry standard for VLSI chip design layouts.

The output of Sponge Paint can be checked using various simulators. For example, we can compare the schematic of the design with the extracted schematic from a Sponge Paint output, or we can check the output by running the layout through a simulator.

2.4.3 Design Rules

There are many different technologies in manufacturing chips and each has its own set of constraints such as the minimum widths of a *metal2* wire. These rules should not affect the overall design of a layout. With the Sponge Paint abstractions, users

do not have to worry about specific manufacturing constraints.

The abstraction interface allows users to set the type of technology they are targeting and accesses the rules appropriately. If a technology is unknown to Sponge Paint, the user is expected to provide the constraints. This flexibility ensures that Sponge Paint will be expandable in the future as the industry undergoes rapid change.

Chapter 3

Sponge Paint Datapath Generator

We discussed the design goals for Sponge Paint in the last chapter. In designing the datapath generator to fit those design goals, we ran into several problems. In this section, we will first describe the datapath, the problems involved in generating a datapath, how we solved them and then our final design.

3.1 How Datapaths Fit into VLSI Design

The layout of a typical VLSI chip can be broken down into several categories:

1. Control logic. This orchestrates the information flow in the chip.
2. Datapath. This is the computation engine. When control logic decides to perform an arithmetic operation, the datapath is the part of the chip that executes the calculation.
3. Memory arrays. This includes RAMs & ROMs.

Datapath is part of the VLSI circuit design that benefits greatly from “... the structured design principles of hierarchy, regularity, modularity, and locality” [10]. Control logic is usually the smallest but the most complicated component of a chip. Its constituents tend to be irregular and are often generated using automatic synthesis

place and route. The memory arrays often take up the most area on a chip, but are highly regular and existing procedural generators work well with regular arrays.

Datapaths are the second largest components on the chip. Their construction is more complicated than the cache but more regular than the control logic. Datapaths are traditionally handcrafted because they have a regular structure compared with control logic such that manual design gives a high layout density. Datapaths consist of components such as high-performance adders, barrel-shifters, and ALUs, that are required to operate at high speeds. As the datapath size is increased, for example with the trend towards more complex microprocessors, design has required enormous layout effort. As K. Usami, *et al* cite, "... new design approach is needed for the large-scale datapath, which reduces the design effort maintaining design quality".

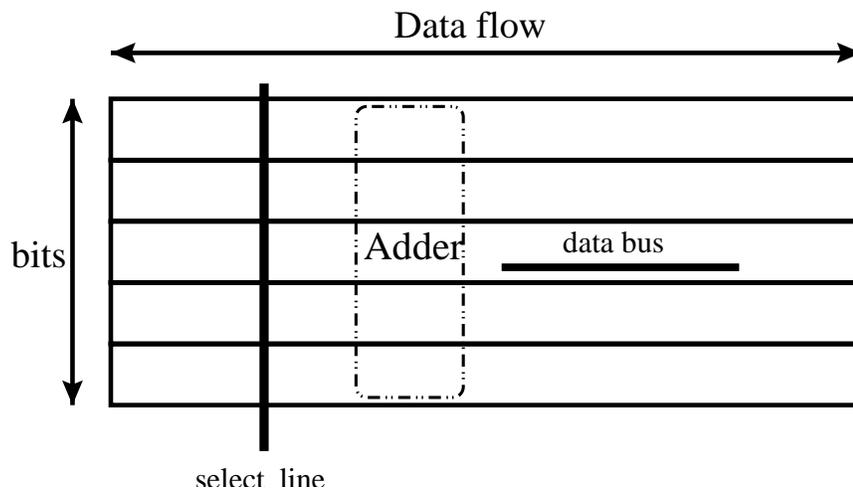


Figure 3-1: Representation of a datapath.

Datapaths usually process n -bit data, which naturally leads to using n identical circuits to implement the structure. Also, data operations can be sequenced in time or space, which leads to placing connecting data operators next to each other. Such properties about the datapath makes it an excellent test ground for Sponge Paint. The regularity exhibited in datapaths makes them ideal candidates for repetitive procedures in generators (See Figure 3-1).

3.2 Design Problems

The main difficulties in the layout of a datapath are:

1. Bit-slices are often regular structures, but some irregularity may creep in for various reasons. Dealing with these irregularities while providing simple and efficient ways to layout the regular parts is a challenge.
2. Bus positions depend on global configurations, and also have to accommodate irregularities while optimizing for regularities.
3. The size and position of a cell not only depends on the size of its contents, but also the number of buses that run through the cell and size of the cells in the same bit slice.
4. Cell terminals have to be wired up to arbitrary bus positions.
5. Overall layout is difficult because of black box inclusions. We have to avoid conflicts between wires laid by higher level builders and the wires in black box components.

3.3 Previous work

Most existing procedural layout programs are process independent because designing a generator is an expensive and complicated task [8]. They often use a “tiler and leaf cells” structure, where a set of basic cells are used as tiles in the main canvas. This

results in high densities and good electrical performance but the resulting schematic is often too complicated and the number of leaf cells too large [4]. This structure is also inflexible and cannot cope with irregularities. In this section, we will describe several other datapath generators and briefly compare them with Sponge Paint.

K. Usami, et al built a datapath generator by stacking a number of identical bit-slices to create a datapath. Similar to Sponge Paint’s motivation, they also emphasize that the designer does “... not need to care about arrangement nor connection for transistors” [5]. Their stacking method, while faster and simpler than Sponge Paint, does not allow connections between bit-slices. Connections between bit-slices are essential for structures such as shifters. Sponge Paint takes longer to generate because each bit-slice is done separately. Even though Sponge Paint is slower for structures that are completely regular, it can handle irregularities easily.

M. Taliercio, *et al* use “over the cell routing” in which each cell has a maximum of eight bus tracks [6]. Sponge Paint does not have this restriction; however, our flexibility sacrifices some potential for optimization. The leaf cells of Taliercio’s system have a fixed height. We allow variable width and height for all the leaf cells. Sponge Paint may be more efficient in cases that use smaller leaf cells since we will adapt to the smaller size.

The previous work that is most similar to Sponge Paint is described by Ben Ammar *et al* in an IEEE paper titled “A High Density Datapath Compiler Mixing Random Logic with Optimized Blocks.” The system they built had three packages: Datapath Global optimizer (DGO), Datapath Layout Generator (DLG) and Datapath Models Generator (DMG).

The DGO is an optimizer for the entire layout. Sponge Paint does not have a similar structure since it allows the users do their own optimized positioning. This decision apparently gives the designer more power. It eliminated much guessing work in developing Sponge Paint. We trust that the designer knows best. Although such an optimizer may be very useful and helpful for minimizing the overall size of the layout by putting components as close to each other as possible after the user decides on the relative positioning of components. The DLG is paralleled with Sponge Paint’s

Grid and performs the same tasks. The DMG is equivalent to the Sponge Paint's component builders.

The Ammar datapath compiler considers components as “leaf cells regardless of their functionalities.” Sponge Paint has the same type of abstraction and views leaf cells as black boxes [2]. The Ammar datapath compiler keeps its leaf-cells in a static library and has a non-static library that generates leaf-cells on the fly. Sponge Paint currently relies on leaf-cells given by the user, although the Sponge Paint primitives are capable of generating leaf-cells.

The Ammar datapath compiler used a specialized language called STYX¹. Sponge Paint is more flexible and universal since it uses a general purpose language. In both Sponge Paint and the Compiler, data busses are horizontal and the control lines are vertical. The Ammar datapath compiler has a set maximum number of 10 bus tracks per bit-slice. This finite number “... may be a limitation of the bit-sliced structure” [2]. The meaning of one bit-slice is arbitrary in Sponge Paint. There is no restriction on the height or width of cells.

The Ammar datapath compiler is similar to Sponge Paint because input and output terminals of the Ammar datapath compiler are “not fixed on the cell abutment box like in classical standard cells.” Sponge Paint does not use abutment boxes, but allows the users to decide the positions of I/O terminals. The differences between the Ammar datapath compiler and Sponge Paint [2] are:

1. The Ammar datapath compiler generates a number of models besides the layout. Although these models are helpful, the majority of them can be generated by specialized programs with the layout as their input. Sponge Paint output includes only the layout.
2. The Ammar datapath compiler uses a symbolic-to-real translation tool such that mapping to a specific process is fully automatic. Sponge Paint performs the translation itself.

¹STYX is the UNICAD Procedural Language, based onto the “Le-Lisp” language

3. The Ammar datapath compiler requires a technology file containing “primitive library and process parameters” for each process. Sponge Paint assumes that the user has access to such a file for the corresponding technology and does not directly deal with the file contents. Instead, Sponge Paint provides a common interface for users to represent design rules.

In summary, Sponge Paint is very flexible compared to what has previously been implemented in datapath generators. Sponge Paint’s flexibility comes from its general purpose language for implementation and also the virtual Grid structure which does not impose any restrictions on the actual physical model or underlying manufacture rules.

3.4 Datapath Layout Generator Overview

The layout style we use for Sponge Paint datapath has horizontal *metal1* wires, vertical *metal2* and horizontal *metal3* wires etc. Figure 3-2 shows an example of a layout using this wiring system. The vertical *metal2* wires such as ground and power lines allow Sponge Paint to have variable height bit-slices, since those lines can be stretched to connect together.

The typical process involved in generating a layout starts with the designer choosing his builder(s). The next step is to inform builder(s) to allocate certain blocks of cells in the Grid. The cell blocks allocated by each of the builders determine their order relative to each other. Each builder has implicit knowledge about the component it builds. For example, a mux builder would take an argument for the bit width of the mux inputs, and place a “base” mux into each grid cell that it allocates. A base mux takes 2 bits as input and has one output bit (See Figure 3-3). Thus, for a 4-bit 2-input mux, 4 base muxes are placed in the grid (See Figure 3-4).

Once all the builders have been chosen and placed, the designer lays down buses to carry outputs and pseudo-buses to connect outputs to inputs. When everything is added, the user commits the design and Sponge Paint outputs a text file in Magic

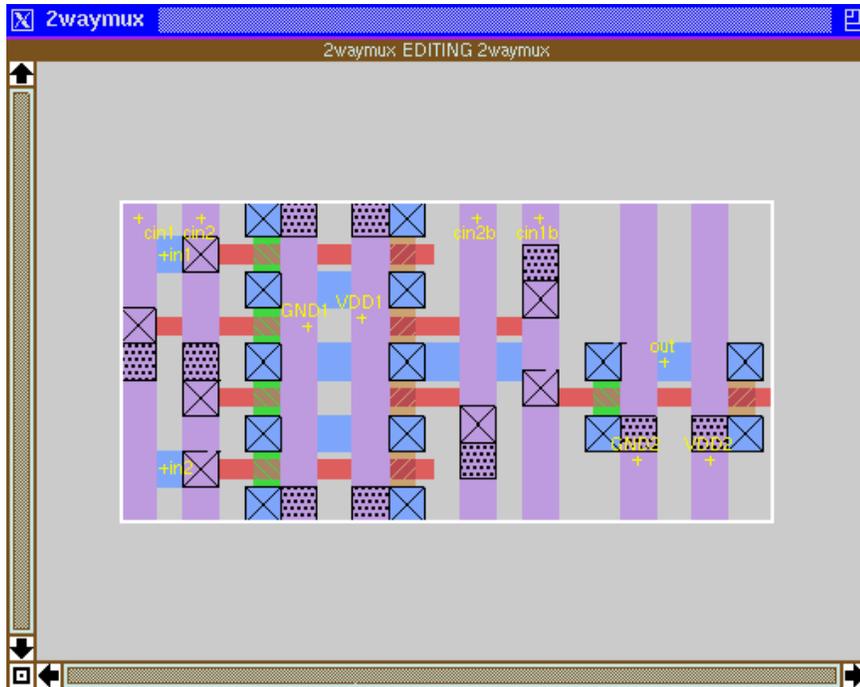


Figure 3-2: The layout of a 1-bit 2-input mux.

format [3] (See Figure 3-5). In the following sections, we will focus on the grid, builders and the interactions between them.

3.5 The Grid

The Grid coordinates with component builders to prevent wiring overlaps. The grid is also responsible for global elements such as buses, and abstracts away exact coordinates of leaf cells. It keeps track of what spaces are still available, what's been used, and which component owns the used space.

The grid is a two dimensional matrix of virtual grid cells. The bottom left corner

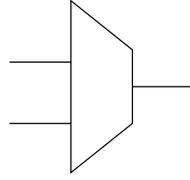


Figure 3-3: A 1-bit 2-input mux.

of the grid maps to the origin: $(0, 0)$. Grid columns and rows that do not contain any contents have the size of zero when the grid is translated into real coordinates (See Figure 3-6).

Each grid row and column can be of any size independent of one another. Both rows and columns automatically scale to the minimum size needed to fit all the contents. For columns, the smallest possible width is the width of its widest component. For rows, the smallest possible width is the larger of the height of the tallest content and the height needed to fit all the buses in the row. This reflex action allows the user not to worry about fitting contents into their rows/columns and increases Sponge Paint's user-friendliness. The variable height rows and variable width columns also provide tremendous flexibility that help to accommodate irregularities in the datapath.

The main difference between grid rows and columns is that rows have the added ability to manage buses. Rows need this capability because buses are horizontal for the most part and individual rows often contain multiple buses. A bus manager keeps track of how many buses there are in a row and how long each bus is. The manager also decides where to assign each bus within a row which subsequently dictates the number of bus tracks needed. The number of bus tracks mandates the minimum height needed to fit all the buses. Thus, bus management plays an important role in minimizing the datapath area as it allows a maximum number of buses in a row before having to increase the height of the row to accommodate its buses.

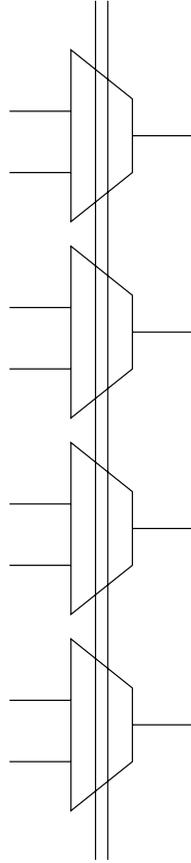


Figure 3-4: A 4-bit 2-input mux.

3.5.1 Grid Border

The border provides a convenient way to add extra space on the sides of the grid to extend special lines such as power, ground and select lines. A border can be added to the grid. A grid border is a ring of rows and columns on the four outer edges of the grid. Because the exact size of the grid and the final number of rows/columns is unknown while the user is still building, the border is a separate entity to be merged with the grid later as a part of the scaling process for rows and columns. The algorithm used for merging the border and the grid is very simple. All the cells in the grid are shifted toward the top right corner by one grid cell. Then the edges of the border are added in to preserve $(0, 0)$ as the origin.

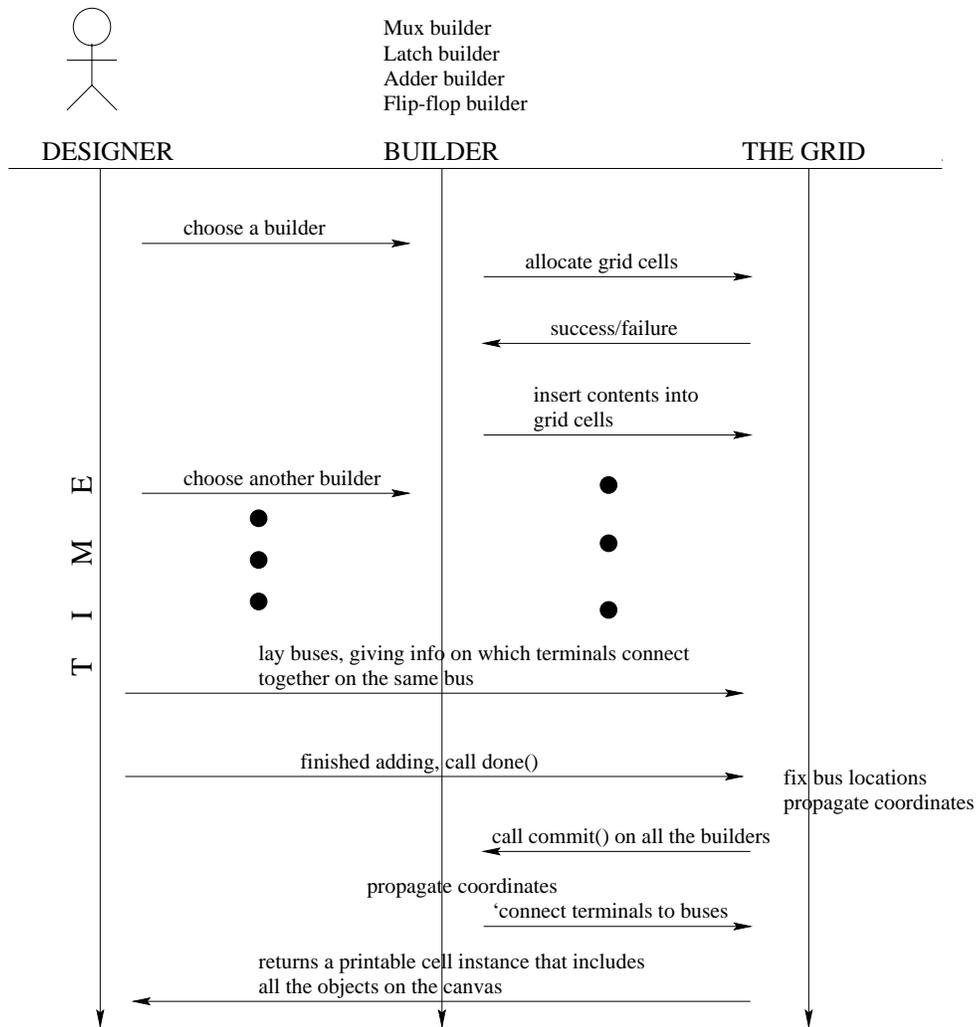


Figure 3-5: Control flow of a layout design using Sponge Paint.

3.6 Builders

Using builders, users place contents in the grid cells. The relative positions of grid cells is a natural representation for the relative locations of user components. A builder communicates with the grid to allocate space for its contents. While the grid mediates between builders, each builder takes care of connections in their own components including children builders if there are any.

All builder objects implement the `SP_Builder` interface. They have the knowledge to layout specific types of components. Builders can be layered on top of other builders to produce more intricate and bigger layouts. In these hierarchies, the parent builder

(3, 0)	(3, 1)	4-bit mux (3, 2)	(3, 3)	
2-bitmux (2, 0)	(2, 1)	4-bit mux (2, 2)	(2, 3)	
2-bitmux (1, 0)	(1, 1)	(1, 2)	1-bit latch (1, 3)	
Cell (0, 0)	(0, 1)	(0, 2)	1-bit latch (0, 3)	

Figure 3-6: A Sample Grid and its Contents

looks upon its children builders as components that need to be connected.

To support the layered approach, builders have notions of an “owner”. By default, the owner of a builder is itself, but when a builder uses other builders, the parent builder becomes the owner of the children builders. The purpose for the owner hierarchy is to allow the grid to call the parent builder to commit positions and connections. The parent builder can decide what to do with the grid’s notification. Most parent builders iterate through the children builders and call the commit position and connection of the children builders.

The majority of builders add leaf cells into the grid. For example, a latch builder would use its leaf cell, which is a latch, to build n -bit latches by stacking n cells and connecting the clock lines. Even though Sponge Paint is capable of building logically more complicated components from simpler ones, it refrains from doing so because users have the best knowledge about their components and thus should be those designing anything involving logic. In the case of mux builders, Sponge Paint does not build 4-way muxes out of 2-way muxes because doing so would require some assumption on how 4-way muxes are built. The user would be unable to add optimization or modifications into the 4-way mux if the Sponge Paint mux builder generated everything. To maximize flexibility, the mux builder expects the user to specify n -input muxes to stack in a similar fashion as the latches. By relying on user for the layout design, Sponge Paint eliminates the repetitive work of copying a

component n times and preserves the user's power to create components as they like.

3.6.1 How To Build a Builder

Since the designer has the best knowledge about his components and its constraints, it makes much more sense to let the designer create the builder that will build the component instead of guessing about what will be desirable and what will not. This attempt at generality is often the downfall of many purely automatic processes. Sponge Paint tries to give the user as much power as possible, while offering shortcuts for the most used operations.

The main goal for the builders was to provide a broad tool base so that users will only need to worry about the behavior of the builder with the grid. The builder is expected to support several behaviors:

buildAt()

is a procedure called by the designer to build in the area given as parameters. The builder can do whatever it needs in terms of what and how much to put into each of the grid cells.

The typical builder builds in one grid column, i.e. it lays out its components vertically. For example, the 32-bit latch is 32 latches connected together by their clocks signals. The vertical configuration is very natural and is also an industry standard. There are exceptions to the “vertically stacked leaf cell” configuration, such as the bus ripper builder. Nevertheless, vertically stacked leaf cells are the most often used and is the default action.

commitPosition()

is called by the grid when the grid rows and columns are done scaling. The builder is expected to adjust the positions of its contents. In most cases, this involves shifting contents into their grid cell boundaries.

commitConnection()

is called by the grid when the buses have been assigned exact coordinate locations. The builder is expected to connect to the buses and accomplish whatever else it needs to make itself a fully connected component.

connect()

connects the given terminal to the given target. The typical builder would just use the default connect, which makes sure that terminals that connect together share a bus. The default connection routine uses a simple algorithm that minimizes the number of turns in the connection. The default algorithm is not fool-proof and will sometimes cause conflicts depending on the type of connections and the cells involved. The designer may choose to make more elaborate connect functions to take advantage of the particular leaf cell's anatomy.

connectTerminals()

is mostly called by builders themselves for convenience. The default action connects a list of terminals vertically and creates a `SP_Label` object to label the entire connection with the given text. This method is where terminals such as the clock signal of latches, all the ground and power terminals are connected together and labeled as such.

other miscellaneous

functions are all implemented by default because most builders have the same action for these methods.

1. `getName` and `setName` for getting and setting the name of the builder, respectively. The builder name can be set by the user to keep track of Sponge Paint builders with corresponding components on the user schematic. `getOwner` and `setOwner` for getting and setting the owner of the builder.

2. `getTerminal` for a particular terminal in the builder contents, `getCellTerminal` for a terminal in a grid cell that belongs to this builder, `getAllCellTerminal` for all the terminals in a grid cell.
3. `getBuilderBox` for the area of this builder in real coordinates, `getGridBox` for the area of this builder in terms of grid cells.
4. `shiftBox` for shifting the area owned by this builder. This method is mostly used by the grid when it has to shift cells while merging with the grid border. Users should not need to use this function because the layout design is known in advance.

The `SP.Basic.Builder` class implements the default functions for everything mentioned above. It is a tremendous help in our process of constructing builders for the datapath generator. Most of the functions have been used for their default behavior. The function that gets modified the most is `commitConnection`.

3.6.2 A special builder: Bus Ripper

Occasionally, the leaf cell model does not fit the structure of a component. The bus ripper is such a case. A bus ripper is a set of vertical wires that connect to horizontal buses bringing the horizontal buses out for ease of connection (See Figure 4-4). As an easy way of telling the builder where to start ripping, the user puts a terminal at the end of the bus, or wherever the user would like to start ripping. The ripper builder takes these terminals as inputs and generates vertical wires and any connection needed to bring the terminal in contact with the vertical wires.

Unlike typical builders, the ripper does not have a leaf cell to tile. As a result, the ripper's `commitPosition` is required to do absolutely nothing because there is nothing for it to move or shift. The lack of leaf cells also poses a problem. If a grid row/column does not have any contents when scaling occurs, that particular row/column will have zero size, i.e. width of 0 for columns and height of 0 for rows. To get around this problem, the ripper builder uses an empty rectangle as its dummy

leaf cell. This rectangle only has size, but no layer or other attributes, so it will not be included in the generated layout. When `commitConnection` is called, the ripper builder's column will be of appropriate size for building the vertical wires.

The `commitConnection` method of the ripper builder simply iterates through all the terminals that it has to rip and calculates the positions for vertical wires.

3.7 Buses

The bus is one of the more complicated components of a layout. There are many factors involving a bus that would have a large impact on the efficiency of a layout. Optimizing the position of buses can minimize lengths of wires used, the relative positions of buses can determine how wide or long the container component has to be and how compact the overall layout will be.

In Sponge Paint, buses are represented by the `SP_Bus` class. The `SP_Bus` class contains the terminals that connect to the bus and the track that the bus is assigned to. Main operations involving buses are done using the helper class `SP_Bus_Pack`, which contains various information about a bus such as the end points of the bus in terms of grid cells, the end points of the bus in real coordinates, and the length of the bus.

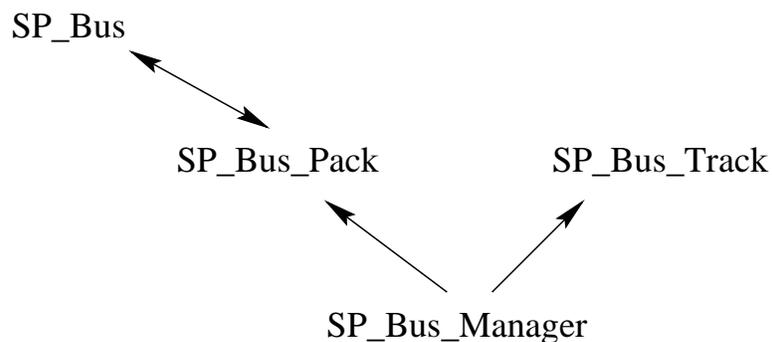


Figure 3-7: Class Dependence Diagram for Bus Related Classes

The `SP_Bus_Manager` class fits buses within a grid row to achieve the minimum number of bus tracks needed. Each bus manager has a list of bus tracks which are horizontal slices within the managed row. The bus tracks provide vertical difference

between the buses. Within each bus track, only the horizontal position of a bus is considered.

The algorithm used by the bus manager to minimize the number of bus tracks is a greedy algorithm – lay the longest bus at the smallest available place into which it would fit.

Terminals can also be extended to places such as the bottom or top of the layout that would not be known until the whole layout is done. The `SP_Extension_Terminal` class is an abstraction so that users do not need to be concerned with the exact size and dimension of their layouts.

The `SP_Extension_Terminal` class is intended for horizontal bus extensions only. Vertical extensions are done separately because vertical extensions will cut through builders and thus have high potential for conflicts. However, the horizontal bus extensions will not conflict with builder contents because buses are on *metal3*, which is not used by ordinary components in development. The front extensions and back extensions are distinguished by a parameter stating whether or not the extension is to the end of the layout or not.

3.7.1 Bus layout

The buses were difficult to layout because they have many constraints. The terminals that the buses connect dictate their horizontal span and thickness. We had initially laid out buses after the grid rows and columns have scaled their coordinates. This approach allowed Sponge Paint to obtain accurate lengths of buses and made optimization easier. However, in order to guarantee that each bit slice had enough space for its buses, we assumed the worst case that each bus overlapped every other bus and thus required the maximum height.

Because datapaths tend to be long rather than high, height changes affect the overall area more than length changes. Since vertical space in datapath is more valuable than horizontal space, we decided to minimize vertical height while sacrificing horizontal space.

Instead of waiting for real coordinates to propagate, we chose to fit buses together

before grid rows and columns were scaled to real coordinates. As a result, we did not know the exact locations of the terminals to be connected. We only knew the virtual grid cell of these terminals. Thus, assuming worst case, bus spans had to cover the entirety of all the grid cells between the end terminals, including the grid cells that these end terminals were in.

This method guaranteed that the bus would be able to reach all of its connecting terminals. It also caused unnecessary overlapping in some cases. The gain of the new method is that we did manage to cut down the height needed to accommodate the buses and saved a significant amount of area overall.

The algorithm for fitting the buses went through a few refinements as well. At first it was the most simple method of taking a random bus and putting it at the first place that it fits into. The buses and their available spaces were in random order, and a new bus was created even if other buses already connect to one of the terminals. Each pair of connections had their own bus.

There are many available algorithms for optimizing bus scheduling. A graph coloring algorithm would be able to solve this assignment problem. However, since we wanted something that was small and fast, we decided to write a quick algorithm using a simple heuristic. The heuristic we used was to fit the longest of all yet-to-be placed buses into the smallest available space that it could fit into. If nothing fits the bus, then we create a new track and put the bus there. We also added a simple way to make sure that terminals that connect together use the same bus. Combining these two changes, the result was satisfactory.

3.7.2 Conflicting wires

Sponge paint draws a lot of wires and connections in the course of making a layout. A difficult problem we had to deal with was that some of these connections were getting too close or overlapping wires that were in the black boxed leaf cells and other builders. Such a problem was almost impossible to detect within Sponge Paint without making a full analysis of the leaf cell contents, this would not only be tedious but also breaks the black box abstraction.

One of the possible solutions was to have the user write a general *isOk* method that returns whether or not a proposed addition would be okay for that particular leaf cell. It is not enough to simply consult builders near the suggested wire, because such decisions may limit possible positions for other wires later and deadlocks could occur. A huge constraint propagation tree would have to be constructed for each wire and connection. We quickly rejected the idea because it would be too computationally intensive.

After some consideration, we decided to avoid the problem altogether by extending terminals outside the leaf cells and at the same time add padding space around the leaf cells. Extending the terminals with the padding space guarantees that whatever connects to the extended terminals would not be too close to components within the black box leaf cells. The extra extension and padding sacrificed some horizontal spacing in exchange for ease and simplicity of implementation. The increase in execution speed was a beneficial side-effect.

3.8 The *done* process

The previous sections described what the user needs to accomplish to set up the layout in the grid. The first step starts with an empty grid (See Figure 3-8). Then the builders are commanded to add components into the grid (See Figure 3-9). The user's main builder can then lay down virtual connections when all the components are in the grid (See Figure 3-10).

When the user finishes adding to the grid using builders 3.6 and other manual components such as extensions and renaming labels, the grid is notified. Upon this "done" notification, the grid launches into a number of activities, the first of which is to unify with the border if the user created one (See Section 3.5.1). The grid then performs a very crude layout of the buses(See Figure 3-11). In this stage, buses are distributed amongst grid rows and the length of the buses are calculated in terms of grid cells. Such initial information helps to scale the rows and columns in the next step.

In the scaling process, each row and column looks through contents in its cells and finds the maximum height and width respectively that they must accommodate (See Figure 3-12). The rows have an extra task to allocate the buses approximately to find out the number of bus tracks needed and to expand, if needed, such that the height of the row will be tall enough for its contents as well as its buses.

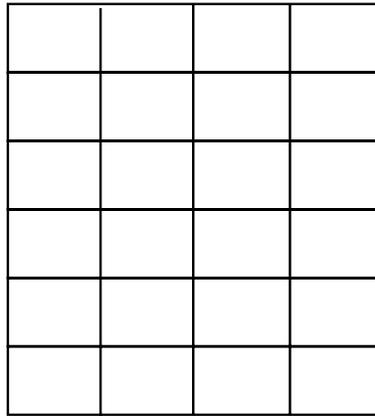


Figure 3-8: Step 1. An Empty Grid

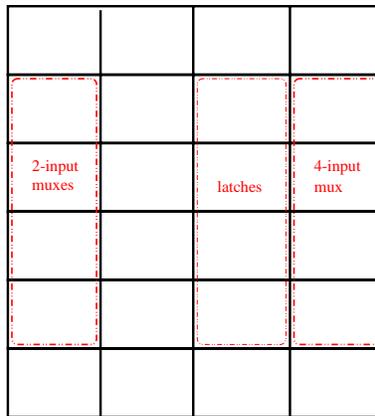


Figure 3-9: Step 2. Add components into the Empty Grid

Once the scaling is done, the grid notifies each of the builders that the coordinates of the grid are known. The builders shift their leaf cells such that they fit into perspective grid cells. The leaf cells were formerly contained in the grid cells by reference only.

Since the rows and columns now have exact coordinates, actual locations are assigned to buses to replace their former symbolic locations (See Figure 3-13). It

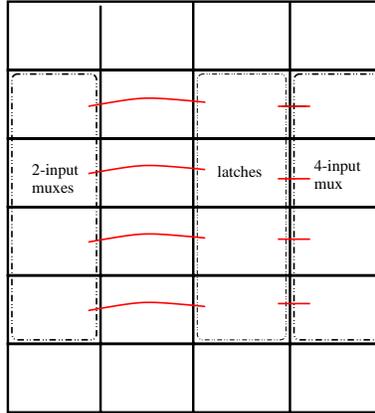


Figure 3-10: Step 3. Add virtual connections between symbolic components.

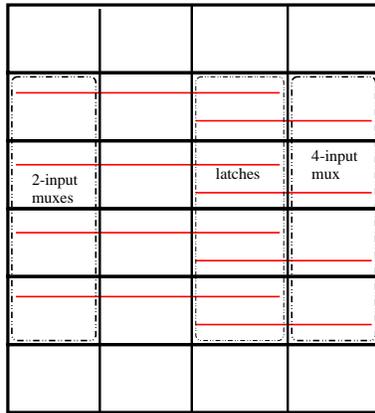


Figure 3-11: Step 4. Crude bus layout after user calls *done()*.

does not matter if this process happens before the builders shift their leaf cells. Since they are independent actions, we arbitrarily chose to shift the grid cells first.

Once the exact locations of buses are know, the builders are notified to make necessary connections to the bus. This is the last step of the actions behind the scenes and Sponge Paint returns a sponge cell object that contains everything in the layout (See Figure 3-14). The user can output the contents as appropriate.

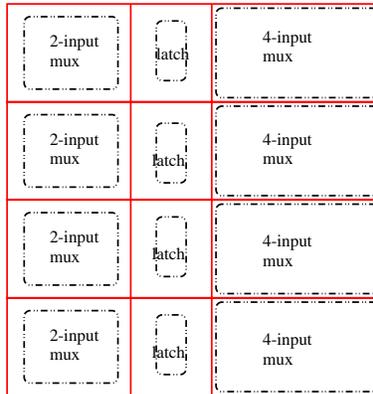


Figure 3-12: Step 5. Grid scaling rows and columns

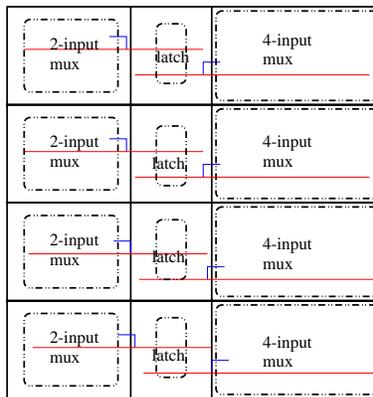


Figure 3-13: Step 6. Exact bus layout done by the Grid and builders connect terminals to the buses

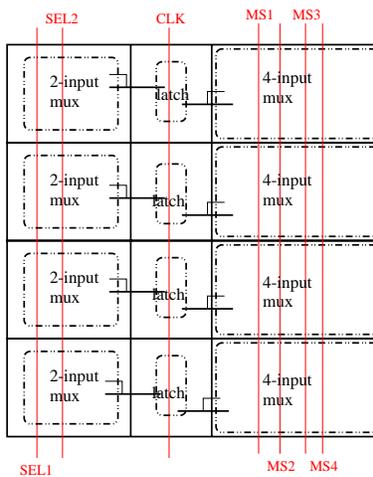


Figure 3-14: Final Step. Include manual non-repetitive components such as select lines and their labels.

Chapter 4

Examples of using Sponge Paint

The Sponge Paint library is organized into several packages (See Figure 4-1). The rest of the packages are relatively self-explanatory. The base package contains the Sponge Paint primitives (Section 2). These are expected to be used over and over again by not only datapath generator but whatever else the users build using Sponge Paint.

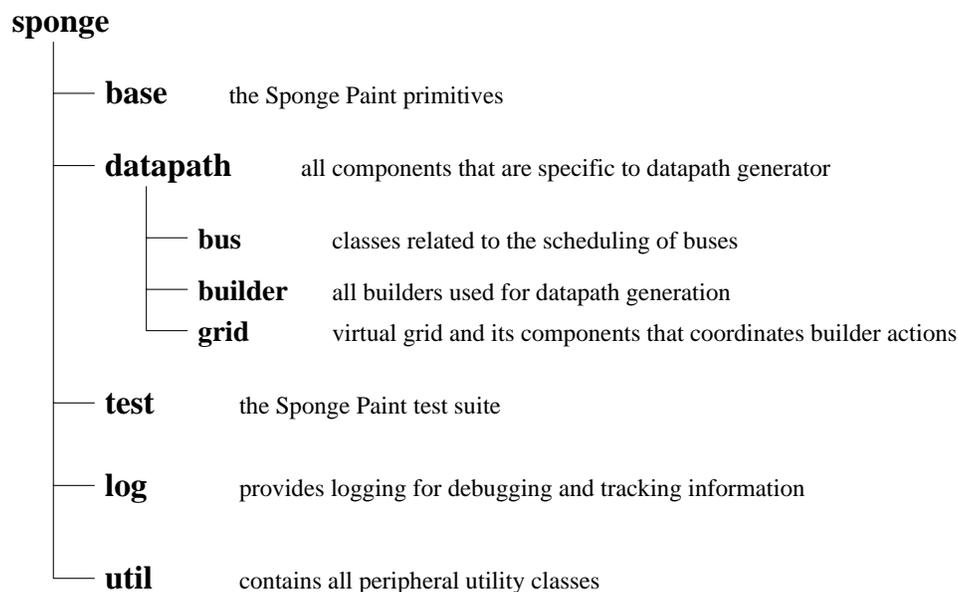


Figure 4-1: Sponge Paint package hierarchy

The utility package contains numerous utilities to make the user's programs easier to write and shorter. For example, a sorting algorithm for sorting bus and available

segments.

We decided to put the `grid`, the `builder` and the `bus` packages under the `datapath` package because they are specifically tailored toward the regular structure of datapaths. Reusing components in the `datapath` package for other components is possible, but the reusable parts will be minor compared to the sizes of the components.

4.1 Datapath Generator

Generation of the CPU bypass network, part of the datapath of a specialized RISC CPU, is the principal example for Sponge Paint. The bypass includes simple builders such as the mux builder and the latch builder intermixed with the Sponge Paint primitives to produce the overall layout.

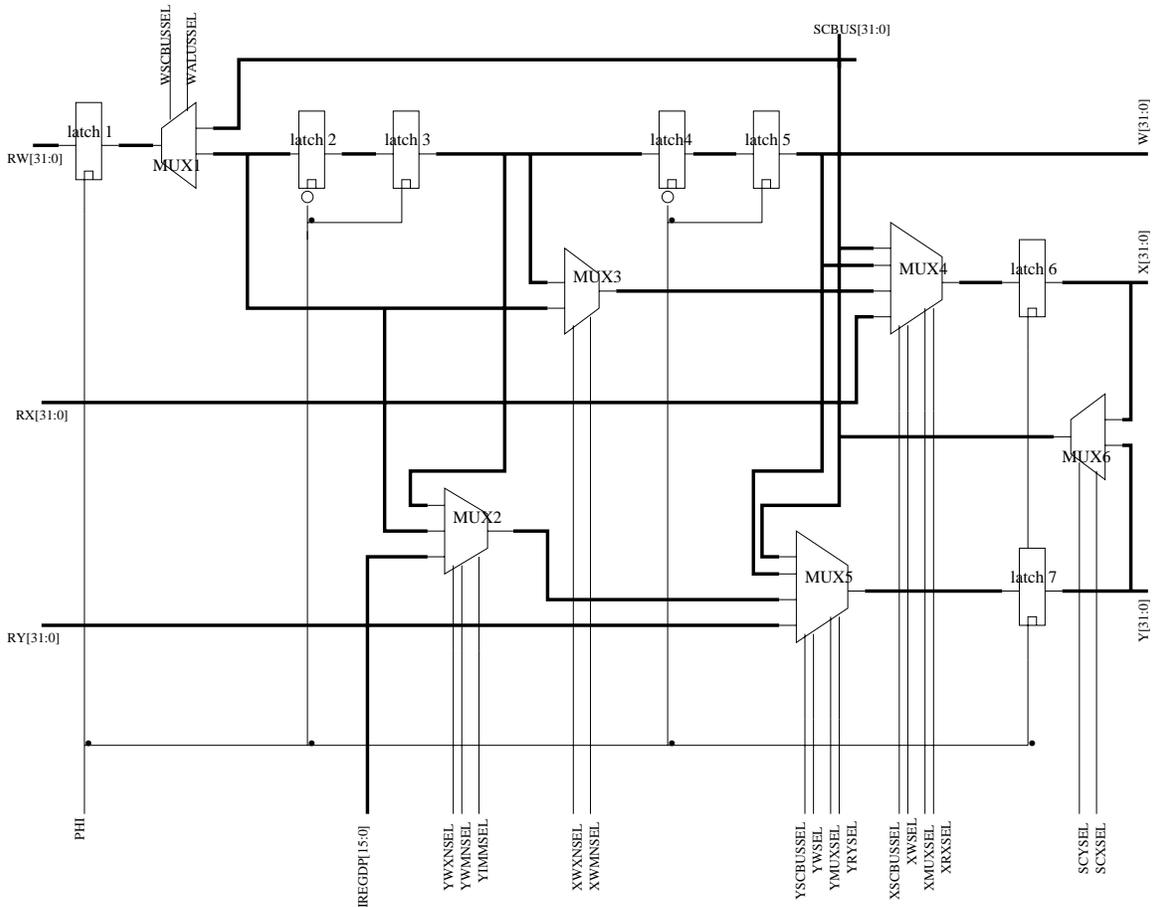


Figure 4-2: A CPU bypass schematic designed by the user.

Builders and the grid are two interdependent entities. The builders put down black box structures for muxes and latches. Primitive components such as rectangles and labels are then used to connect black box structures.

Figure 4-2 is the schematic of the CPU bypass as designed by the user, and Figure 4-3 is the 32-bit CPU bypass generated by Sponge Paint. In order to prevent wiring conflicts, terminals in leaf cells are extended outside the leaf cells on the left and right sides. This simple heuristic allows the terminal connection algorithm to do whatever it needs to reach its bus without conflicting with wires inside the leaf cells. A possible optimization would be to have better connection algorithms for each type of builder.

4.1.1 Implementation

The datapath bypass generator class inherits from the `SP_Basic_Builder` class and becomes the builder that constructs the bypass layout. The bypass builder over-writes a few of the methods in the basic builder:

1. **buildAt()** in the bypass builder initializes an array of child builders such as mux and latch builders, and puts them into the Grid. The bypass builder also adds in virtual connections between all the child builder components.
2. **commitPosition()** in the bypass builder recursively invokes the same method in child builders.
3. **commitConnection()** in the bypass builder recursively invokes the same method in child builders. After all the child builders finish committing connections, the bypass builder renames select lines and extends all the ground and power lines.
4. **shiftBox()** in the bypass builder recursively invokes the same method in child builders.

The `SP_Basic_Builder` class was very helpful in creating new builders. All the basic capabilities were inherited and the user only needed to implement the different behaviors. The bulk of the bypass builder was in the `buildAt()` method where virtual

connections are created. This result is expected because the main work for a bypass builder is to connect its components. The observation further confirms the validity of the builder interface.

4.2 Other examples

4.2.1 Vertical Bus Ripper

All data buses in Sponge Paint are horizontal. The bus ripper allows a user to bring out buses vertically. The ripper builder occupies one vertical column in the grid.

Grid rows and columns that do not have any contents in them will have a size of zero when scaling happens. In order to have a positive size for the column that the ripper builder is in, the builder inserts empty rectangles into the grid. These rectangles only have a size and does not have a layer, which in Magic terms, means that it's empty. The ripper builder incrementally adjusts the size of the inserted rectangle as users add terminals to rip. A bus to be ripped is represented by a terminal at the point where the rip is to be at.

Besides all the functions inherited from the `SP_Basic_Builder` class, the ripper builder has an extra method: `addTerminal()`. The input to the `addTerminal()` method are terminals on target buses to be ripped. The terminals provide physical locations of the buses.

Similar to the bypass builder, the ripper builder also overwrites a few methods:

1. `commitPosition()` in the ripper builder does nothing, because exact locations needed by the ripper builder is not available yet.
2. `commitConnection()` in the ripper builder lays down the bus rips by connecting target bus terminals with wires and vias.

The fundamental difference between the bus ripper builder and typical builders is that the bus ripper builder does not use any standard leaf cells. As Figure 4-4 shows, the bus ripper's column does not have any leaf cells. This example illustrates

the flexibility and abstraction of Sponge Paint very well, as to the Grid, the ripper builder is just like any other builder and exhibits the same behavior through modifying relevant methods.

4.2.2 32-bit Shifter

Shifters are important parts of many microprocessor designs that involve arithmetic shifting, logical shifting and rotation functions. We can build a 32-bit shifter using Sponge Paint. The `SP_Shifter_Builder` is a barrel shifter builder that uses the appropriate number of 2-input mux builders to form the shift stages. The shifter builder connects the stages together. The user provides parameters for the number of bits to shift and the leaf mux cell to use. This flexibility allows the user to build many different kinds of shifters, such as using 4-input muxes with powers of 4 skips between shift columns.

The default extension for left and right sides of a base cell are over-written for shifters, because as the number of skips increase, the space between shift columns has to increase in order to accommodate the number of vertical wires.

The only method in the parent class that is over-written by the shifter builder is `commitConnection()`. In the shifter builder, this method recursively invokes the same method in the mux builders and then connects shift stages. As the Figure 4-5 shows, many connections between stages are vertical and crosses bit-slices. The shifter builder is another illustration of Sponge Paint's flexibility.

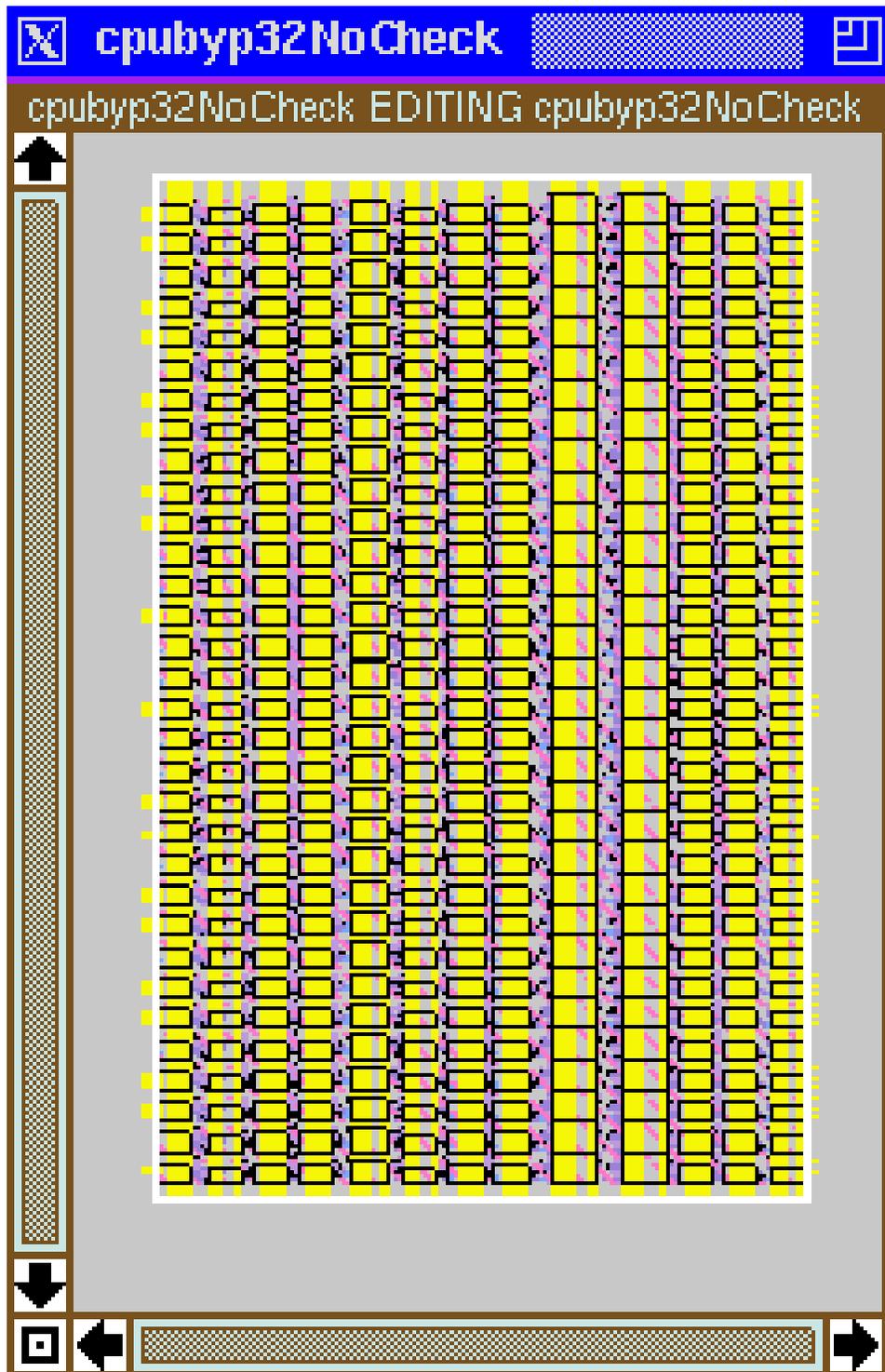


Figure 4-3: A CPU bypass generated by the Sponge Paint datapath generator.

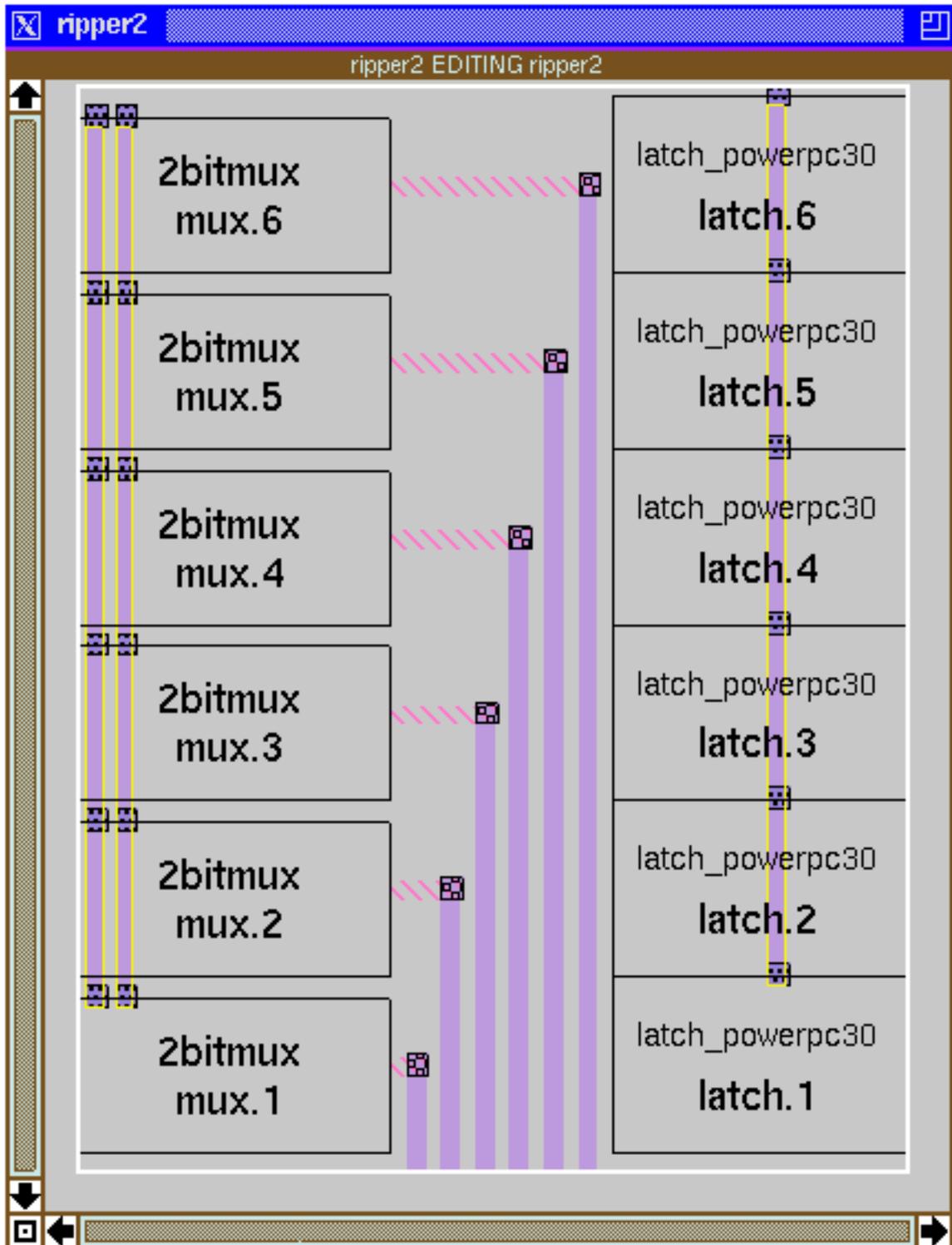


Figure 4-4: The Sample Output of a Bus Ripper Builder.

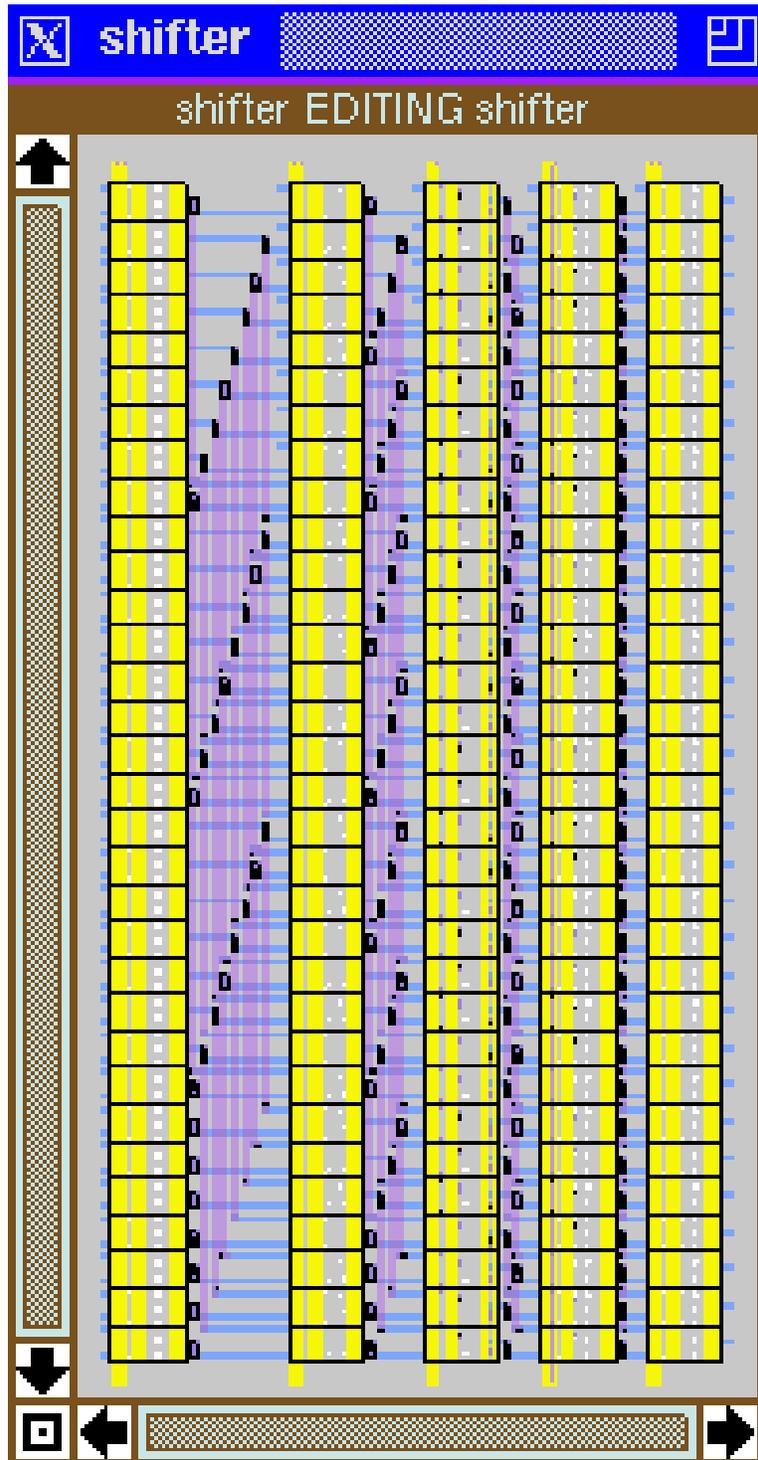


Figure 4-5: A sample 32-bit shifter generated by Sponge Paint.

Chapter 5

Conclusion and Future Improvements

Sponge Paint successfully achieved its goals of flexibility, generality and incremental and hierarchical structure. Sponge Paint takes the tedious details away from the human designer while providing flexibility and extensibility. The user makes the ultimate decision on what goes where and when. Sponge Paint does not perform exceptional optimizations because strict optimization often compromises flexibility. It is difficult to stay flexible while maximizing optimization.

However, users do not need to compromise optimization using Sponge Paint. The designer can easily optimize the way builders route wires, or change the way buses are scheduled. In this case, giving users power results in the best balance. Designers decide what goes in a layout, thus they should have the power to control the automated layout.

The Sponge Paint library is approximately 8,500 lines of Java code. This count includes the datapath generator and supporting classes, utility classes and all other sample classes. The datapath bypass generator was only about 300 lines of code. The entire design and implementation of Sponge Paint took approximately five months, from the end of September of 1999 to mid-March of 2000. I was the only programmer working on Sponge Paint. In terms of software development resources, this is a relatively short amount of time and people-power. We focused on flexibility and

extensibility, which helped us to avoid the “do everything” mentality.

There are many possible extensions for Sponge Paint. Because of its flexibility, Sponge Paint is not restricted to generating datapaths only. For example, it can also be used to generate RAMs and ROMs. Sponge Paint can even generate leaf cells.

5.1 Future Improvements

Generating a 32-bit datapath from a Sponge Paint program takes a long time, about 20 minutes for the bypass network. This time period is very long in computer execution time. Even though Sponge Paint was not designed to optimize running time, it may be beneficial to make it run faster in the future.

There are components in a VLSI chip layout that are more complex than the datapath. If a user were to use Sponge Paint to generate all the components at once, it would take even longer than 20 minutes. A possible cause for this latency is the fact that each bit-slice generates its own bus scheduling. When bit-slices are very similar, it could save time to cache bus assignments.

Time costs are not a large concern if the generation is only done once, however, in the event that the user had to make a small change, it would take another day for the modified layout to finish. Incremental output would solve this problem. The designer should be able to use the Sponge Paint output as input for the next Sponge Paint execution.

For the future, a Verilog¹ frontend can be installed to describe datapath structure. The integration with Verilog would make Sponge Paint much more user-friendly and will make it more appealing to hardware oriented designers who may not be competent Java programmers. Also, other tools may use Verilog as input.

Some other optimizations can be made for Sponge Paint:

1. Builder optimization for terminal-to-bus connections so that wires can go through leaf cells instead of being routed outside as a heuristic to avoid conflicts. This

¹Verilog is a general language describing hardware.

would result in more compact and efficient design.

2. Grid should allow compound builder borders for structures such as pad rings that have a naturally border-like shape.
3. Add in an event model so that the order of events that happen in *done()* can be changed outside the library source code. This allows true modification in the heart of the generator.
4. Monitor performance for statistic analysis.
5. Optimize Sponge Paint to minimize the amount of code a designer must write to generate a design.

We were very pleased with the Sponge Paint library for procedural layouts. It is demonstrably flexible and portable. The abstractions for input, output and manufacture rules have been great time savers in the process of developing the datapath generator and will no doubt save time for future users. Sponge Paint shows great potential to generate production-ready layouts².

²The documentation for the Sponge Paint layout library can be found at <http://www.cag.lcs.mit.edu/spongepaint/index.html>

Appendix A

Programs

A.1 H-tree Java Code

```
/*
 * This class builds an H-tree using Sponge Paint. An H-tree is a
 * fractal branching from a letter H, and has the property that each
 * leaf point is equidistant to the center of the H. It is often used
 * in chip designs to deliver clock signals because each signal would
 * arrive at all the leaf points in the same amount of time.
 *
 * @author Gong Ke Shen
 * @version 2.0, 03/12/2000
 */
package sponge;

import sponge.util.*;
import sponge.base.*;

public class SP_H_Tree {
    public static void main(String[] argv) {
        SP_Pen pen1 = new SP_Pen();
        SP_Cell canvas = pen1.getCanvas();
        int length = 128; // the initial length. Must be at least 2^level.
        int level = 4; // the number of recursions for the H-tree
        int width = 16; // the initial width. Must be at least 2^level.

        // draw the left half of the tree
        drawH(level, pen1, length/2, width, true);
    }
}
```

```

pen1.reset();
pen1.getCoord().shift(new SP_Coord(0, -width-width/2));
pen1.drawRight(length, width);

// draw the right half of the tree after positioning the pen to a
// correct location
SP_Pen pen2 = new SP_Pen(pen1);
pen2.getOrigin().shift(new SP_Coord(0, width/2));
pen2.getCoord().shift(new SP_Coord(0, width/2));
drawH(level, pen2, length/2, width, true);

// output the tree to stdout
SP_Output output = SP_Process.getOutput();
String magFile = output.toString(canvas);
System.out.println(magFile);
}

// the recursive function
public static void drawH(int level, SP_Pen midPen,
int length, int width, boolean upDown) {
    if (level > 0) {
        SP_Pen pen = new SP_Pen(midPen);
        if (width == 0) width++;

        // draw up-down wires
        if (upDown) {
pen.getOrigin().shift(new SP_Coord(0, -width));
pen.getCoord().shift(new SP_Coord(0, -width));
pen.drawUp(length, width);
drawH(level-1, pen, length/2, width/2, false);

pen.reset();
pen.drawDown(length, width);
drawH(level-1, pen, length/2, width/2, false);
        }
        else { // draw left-right wires
pen.getOrigin().shift(new SP_Coord(-width, 0));
pen.getCoord().shift(new SP_Coord(-width, 0));
pen.drawRight(length, width);
drawH(level-1, pen, length/2, width/2, true);

pen.reset();
pen.drawLeft(length, width);
drawH(level-1, pen, length/2, width/2, true);

```

```
}  
  }  
}
```

Bibliography

- [1] "FPGA Design with Xilinx/Mentor Graphics Tools".
- [2] L. Ben Ammar and A. Greiner. "A High Density Datapath Compiler Mixing Random Logic with Optimized Blocks". In *The European Conference on Design Automation*, February 1993.
- [3] W. Scott etc. "Berkeley CAD Tools User's Manual".
- [4] A. Greiner and F. Petrot. "Using C to Write Portable CMOS VLSI Module Generators".
- [5] N. Matsumoto K. Usami, Y. Sugeno and S. Mori etc. "Hierarchical Symbolic Design Methodology for Large-scale Datapaths". In *IEEE Journal of Solid-State Circuits*, pages 381–385, March 1991.
- [6] G. Foletto M. Taliercio and L. Licciardi. "A Procedural Datapath Compiler for VLSI Full Custom Applications". In *IEEE Custom Integrated Circuits Conference.*, pages 22.5.1–22.5.4, 1991.
- [7] Y. Watanabe N. Matsumoto and S. Mori. "Symbolic Design Methodology for High-Density Macro-Cell". In *IEEE Custom Integrated Circuits Conference.*, 1988.
- [8] W. etc. Schardein. "Analog Module Generators for Effective Design Assistance".
- [9] Cadence Design Systems. "The SKILL Reference Manual". San Jose, California, USA, October 1991.

[10] N. Weste and K. Eshraghian. Addison-Wesley Publishing Company, June 1987.