A Datapath Compiler with Technology Portability

by

Jianghong Hu

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

A Datapath Compiler with Technology Portability

Jianghong Hu

Master of Applied Science, 2000

Graduate Department of Electrical and Computer Engineering

University of Toronto

Abstract

In digital signal processing (DSP) ICs and microprocessors, the datapath is the core where all computations are performed. A datapath compiler is generally used to automatically generate layouts for datapaths by taking advantage of the bit-sliced structure presented in datapaths. Conventional datapath compilers focus on using regularity to generate layouts with minimum area, while the timing issues are not well addressed. Some attempts are being made to attack this problem in synthesis tools, which employ logic optimization techniques to synthesize datapaths in a regularity-aware manner, and then generate netlists with placement data to guide the physical design. We approach this problem from the opposite direction, i.e., we start from physical design based on module generation and then plan to incorporate timing optimization techniques into the system in the future. The work presented in the thesis is the first step towards this direction.

A datapath compiler has been developed and implemented based on an object-oriented design that allows the future extension for static timing analysis and optimization. The timing issues have not been addressed directly in the thesis. A set of Java classes has been developed for design capture. A new type of over-cell routing algorithm designed specifically for datapaths has been proposed and implemented. This is the first time that a cell layout generation tool has been used to generate cells on the fly in a datapath compiler. Using CLASSIC-SC for cell generation is also the main reason that we can achieve technology portability. Other features of the datapath compiler include parameterization in bit width, the ability to deal with the irregularity that exists in real-world datapaths, and a GUI for capturing designs, displaying and exporting layouts.

A comparison with a custom layout shows that this new datapath compiler can produce comparable layouts in area while being technology portable.

Acknowledgments

I would like to express my gratitude to my supervisor, Professor Paul Chow, for his guidance and help. His advice and support have made my graduate study a happy learning experience. Most of all, I am truly grateful to him for his understanding and encouragement. I would also like to thank Professor Zvonko Vranesic, Glenn Gulak, and Jonathan Rose, who have given me help during my graduate study, for their valuable time and suggestions.

This thesis would not have been possible without CLASSIC-SC, an automatic cell layout generation tool from Cadabra Design Libraries Inc. The flexibility that CLASSIC-SC provides is the source of most ideas in the thesis. Thanks to Cadabra Design Libraries Inc. for letting us use this tool in our research. I would like to thank Peter Wilson for providing technical support.

Special thanks to Sean Peng and Louis Zhang for their tremendous help and encouragement throughout the past two years. I am also thankful to them and Qiang Wang, Brent, Edward, Franklin, Shuo, Anwei, Jinheng, Zhixian Jiao, Wei Yang, Song Ye, Yucai Zhang, and Jianhui for all the good times, technical discussions, and a friendly research environment.

I am very grateful to my parents and my husband Xiaojun for their love, understanding, patience and encouragement.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation

In digital signal processing (DSP) ICs and microprocessors, the datapath is the core where all computations are performed. A system's performance is largely determined by the design and implementation of its datapath. A typical datapath consists of N-bit wide operators and memory elements interconnected by a bus-oriented structure. The regular bit-sliced structure of the datapath makes its implementation a special topic in the VLSI physical design field.

Because of the important role they play and topological regularity they possess, datapaths were implemented in a hand crafted manner in traditional VLSI design. But as designs became complex so did the datapaths. In the late 1980s and early 1990s, great efforts were made to automatically generate the layout for datapaths [Amm93, Cai90, Coh91, Gor89, Tal91]. Datapath compilers based on module generation techniques rose as a cornerstone at that moment. They produced dense layouts in a fraction of the manual design time without compromise on area. However, in today's ASIC design, datapaths are seldom being designed and implemented specifically as a separate block. Instead, they are being logically optimized, placed and routed like random logic, together with other parts of the circuits such as control logic. Datapath compilers have been neglected on most occasions. One reason for this is that the synthesis-based design methodology using standard cells is the mainstream in today's ASIC design; the other is that datapath compilers are hard to fit into the current design flow that is formed by widely used automatic design tools. The latter occurs mainly because conventional datapath compilers only focus on using regularity to generate layouts with minimum area, while timing issues have not

been well addressed [Len98]. To date, some attempts have been made to attack this problem in the synthesis tool [Hai97], which employs state-of-the-art logic optimization techniques to synthesize datapaths in a regularity-aware manner, and then generates netlists with placement data to feed into the downstream physical design. In this thesis, we decide to approach this problem from the opposite direction. More specifically, we start from the physical design that is based on module generation, and then, the techniques used for logic synthesis can be incorporated into the module generator for optimization. We think this strategy is appropriate for datapath design. A trend to converge logic synthesis and physical design in today's Electronic Design Automation (EDA) industry also encourages us to do so — the existing partitioning of the design flow as logic synthesis and physical design is to handle the complexity, while the shrinking time-to-market window requires fewer iterations between these two phases [Keu97]. The inherent topological regularity in the datapath suggests that, within manageable complexity, it is possible to combine the timing optimization, even logic optimization, with physical design into a mixed software package. The thesis is the first step towards this direction.

The extensive use of pre-designed and pre-characterized blocks, or Intellectual Property (IP) cores, to design complex integrated circuits today is another reason for us to adopt the module generation approach [Rin97]. One challenge facing core-based design is that most IP cores are still unable to offer the technology portability and expected performance at the same time. On the one hand, HDL-based soft cores are technology portable but without guarantees in performance, and predictability in physical dimension; on the other hand, firm cores captured in gate-level netlists and hard cores delivered as layout entities provide advertised performance but without the flexibility in target technology [Gup97]. We believe, for datapaths, module generation is a solution to this problem. By using module generation techniques, we have the flexibility in one piece of software to control the technology portability, and to predict and manage the topological and electrical characteristics. It is said that the above-stated restrictions are partly caused by the format used to deliver the core design: VHDL and Verilog, which are both function-oriented languages. In this thesis, we propose an object-oriented framework with a set of Java APIs that can be used to capture datapath designs in textual descriptions at a structural level. This framework is the basis for our datapath compiler to be built on and to be expanded.

The datapath compiler we present in this thesis is cell-based. It needs a large number of cells as the support. The availability of a commercial tool, CLASSIC-SC from Cadabra Design Libraries Inc. [Cad00], for cell generation makes this research possible. In traditional datapath compilers, the cells that comprise datapaths are designed specifically and saved in cell libraries; while in our approach, we use CLASSIC-SC to generate cells on the fly. CLASSIC-SC is a flexible system that allows us to control its behavior to generate cells for our special needs.

1.2 Research Approach and Objectives

In this thesis, a parameterized datapath compiler is implemented using a cell-based module generation approach based on object-oriented design. An object-oriented framework is created. A datapath (a module) is designed by writing a subclass of classes from the framework; the layout of the datapath is obtained by instantiating an object from the subclass.

As we will show in the next chapter, traditional datapath compilers have several major problems. First, most datapath compilers neglect the irregularities that exist in the real-world datapaths because regularity is always emphasized when considering the physical implementation of the datapaths. Second, all of the datapath compilers rely on the libraries that contain predesigned, and precharacterized cells and blocks representing the layouts of all possible units. However, the use of libraries is the main obstacle to achieving technology portability, which is the ability of retargeting a design to other process technologies. Retargeting needs to reproduce all the elements in libraries, most often by hand. In addition, the capacity of libraries largely affects the quality of final layouts. The finite varieties of a cell with different heights, and hence the fixed number of tracks available for routing in each cell, can be a serious limitation for complex datapath designs. Third, all the datapath compilers in published research were written using procedural languages. In fact, procedural languages are not well-suited for module generation techniques for the following two reasons. First, procedural languages tend to obscure the geometrical relationships such as the relative locations of the cells, which are always needed in module generation techniques [Gaj88]. Second, flexibility and extensibility are indispensable characteristics for a

datapath compiler to deal with different kinds of datapath designs. These characteristics can be more easily achieved in the software based on object-oriented design than those based on the procedure-oriented design implemented using procedural languages.

In our approach, the first problem is addressed by allowing some random logic mixed in the bit-sliced structure. The second problem is resolved by employing CLASSIC-SC, a cell generation tool that is technology portable, to generate cells on demand. In theory, routing and optimization tools can request cells of various functions and electrical characteristics without limitation. The third problem is addressed by applying object-oriented design. In the object-oriented framework we create, new cells can be easily added into the framework by writing new subclasses; cells of the same function with different transistor sizing, i.e., different drive strength, can be easily obtained by instantiating corresponding objects from one class. Even though our system does not directly address the timing aspects of the optimization, the object-oriented framework is designed to be extensible, which will enable the addition of timing optimization in the future.

To implement the cell-based datapath compiler, an over-cell routing algorithm has been developed. The whole routing phase is completed by employing a mixed over-cell and channel router.

The objective of the thesis is to develop a datapath compiler, with the following goals:

- To provide a set of APIs to capture the datapath design hierarchically at the structural level, with placement directives for users to specify the placement of cells.
- With special attention to design the classes for reuse, and class hierarchy for extension.
- To develop a new type of over-cell routing algorithm aimed at minimizing the area for datapaths; to implement a mixed over-cell and channel router.
- To integrate CLASSIC-SC into the datapath compiler and make the whole system technology portable.
- To make the datapath compiler parameterized so that the bus width could be specified by users when generating the layout.

- To implement a Graphical User Interface for capturing the design and displaying the layout.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 gives background information, including a description of the general approach to datapath layout generation, a brief introduction to a cell generation tool: CLASSIC-SC, a review of previous work on cell-based datapath compilers and routing algorithms.

Chapter 3 presents the datapath compiler we developed in this thesis. The datapath compiler is introduced in terms of its design capture, cell design, placement, routing, and important features. We present a set of Java APIs that are used to capture designs hierarchically. We show how to collaborate with CLASSIC-SC to generate cells for our special use. We propose a new type of over-cell routing algorithm for datapaths. The implementation of a dogleg channel router, which serves as a complement to the whole routing phase, is also described.

The object-oriented design of the datapath compiler is presented in Chapter 4. The structure and function of the primary classes are discussed. The objected-oriented design of the whole system is depicted using Unified Modeling Language (UML) [Rum99], which is a visual modeling language used to specify, visualize, construct, and document the artifacts of a software system, especially the software developed using an object-oriented approach.

Chapter 5 presents important implementation details and results by running some application examples. We show how we make CLASSIC-SC for our use, and how we implement the GUI-based design in Java. We give three typical experimental results of the datapath compiler. The results are compared with those produced from either a standard-cell based synthesis design flow or manual layout.

Chapter 6 summarizes the thesis conclusion and provides directions for future work.

A brief introduction to Unified Modeling Language is given in Appendix A.

A short user manual for our datapath compiler is given in Appendix B.

CHAPTER 2

Background and Previous work

This chapter introduces background information and reviews previous work in the relevant areas. When discussing published research, we will describe in some detail the previous work in routing algorithms because routing for datapaths is a subject we emphasize in this thesis. Section 2.1 describes the general approach to datapath layout generation. Section 2.2 gives a review of previous work on datapath compilers. We focus on the routing algorithm in each system. Section 2.3 describes a channel routing algorithm that we use to implement the channel router in our system, and briefly introduces general-purpose over-cell routing problems. Section 2.4 provides brief background information on a cell generation tool, CLASSIC-SC, which we use in the background to supply the cells needed. Section 2.5 gives a summary.

2.1 General Approach to Cell-based Datapath Compiler

Before presenting the previous work on cell-based datapath compilers, an introduction to the general approach in this area would be useful. This provides the basis not only for the following sections but also for the whole thesis.

By taking advantage of the nature of the regularity that is presented in datapaths, traditional datapath compilers arrange the datapath layout in a bit-sliced fashion, as shown in Figure 2.1(a). In the ideal situation, the regularity is provided in one dimension — an N-bit datapath is constructed by repeating the same slice N times. The cells within a bit slice are of the same height. Data are arranged to flow in one direction, while control signals are introduced in an orthogonal

direction to the data flow.

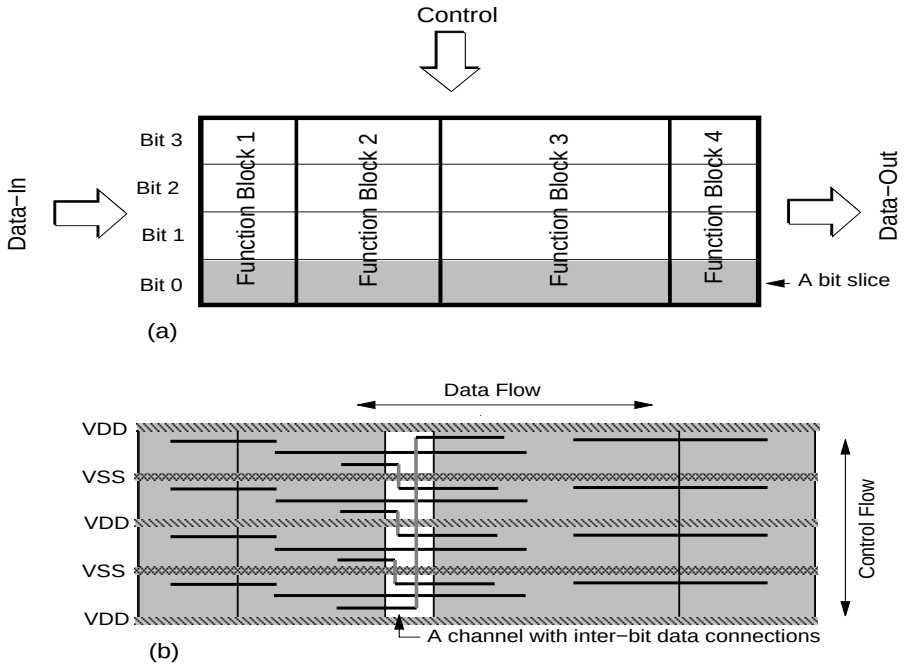


Figure 2.1 *(a) Bit-sliced datapath organization (b) Bit-sliced layout structure*

Figure 2.1(b) shows the layout structure of the datapath in Figure 2.1(a). The entire datapath layout is obtained by routing interconnections between function blocks (FBs), each of which is either an N-bit optimized module (adder, register file, etc.) or a composed module made by stacking N one-bit cells (nor, multiplexer, etc.). FBs are placed horizontally by applying a linear placement, abutting on each other. Neighboring slices are abutted as well. By mirroring the even and odd slices around the horizontal axis, the nWells, V_{DD} rails, and V_{SS} rails can be shared between neighboring slices. To make the abutments possible so as to generate highly dense layouts, over-cell routing methods must be employed to route the interconnections.

However, when using a process technology with a limited number of metal layers, it is not guaranteed that the final layout can be achieved using only abutments. Extra routing areas outside the cells are needed to route some of the connections, especially the inter-bit interconnections, whose pins reside in different slices. For instance, in a technology with two metal layers, cells are mainly constructed using metal 1 and poly; V_{DD} and V_{SS} rails run in metal 2; control signals are routed vertically in metal 1 as cells have been specifically designed to leave the space for this use;

data connections are routed over the cells in metal 2. To connect the inter-bit signals, metal1 is the only choice to stride across the power rails and horizontal data connections, while at the same time it is not allowed to run over the cells. In this situation, channels are needed to finish this part of the routing, as shown in Figure 2.1(b).

When designing a datapath compiler for real-world design, there are two issues we have to take into account. First, although the regularity of the bit-sliced structure is the most distinctive characteristic in a datapath, datapaths are not fully regular in most cases. Figure 2.2 shows the abstract view of a datapath with irregularity. A datapath compiler should be able to deal with a small amount of random logic in the datapath. Second, to make the above-mentioned approach work properly, we must design the cells specifically for over-cell routing rather than using standard cells. However, specifically designed cells always cause a problem in technology portability.

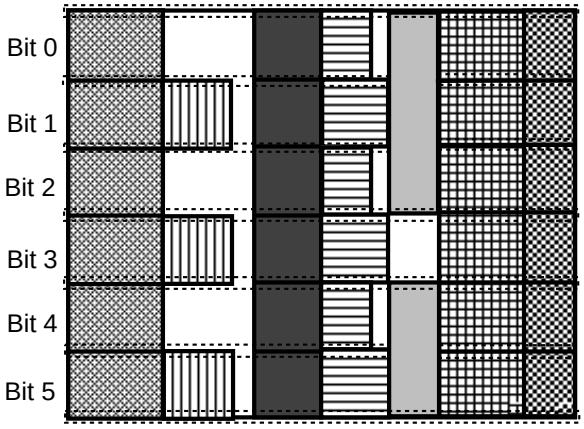


Figure 2.2 *The abstract view of a datapath with irregularity*

In the following section, we introduce three previous cell-based datapath compilers that stem from this general approach.

2.2 Datapath Compiler

This section provides a detailed description of three typical cell-based datapath compilers,

including a datapath layout assembler in CATHEDRAL-III [Cai90], Pathway [Coh91, Coh93], and a datapath compiler that is able to deal with random logic [Amm93]. Each of them is described in terms of its overall structure, cell design, placement strategy, and routing algorithm. It is said that in datapath design placement can always be done manually in an efficient way because data flow provides an obvious indication for placement [Rab96]. We take this argument for granted in the thesis, and hence, make more efforts to develop a routing algorithm for datapaths. As well, in this section, when introducing previous work, we study the corresponding routing algorithm in detail. The three systems are discussed in order of the time when they were published, beginning with the earliest. In Section 2.2.4, a comparison is given to summarize the strengths and weakness of the three systems introduced.

2.2.1 A Datapath Layout Assembler in CATHEDRAL-III

This system is a part of a silicon compilation environment, CATHEDRAL-III, which is designed for high performance DSP circuits. It consists of a linear placement tool, a track assignment tool and detailed routing tools. It takes structural netlists from synthesis tools and generates final layouts as the output. A library serves at the background to provide the function blocks and cells specifically designed for datapath.

The datapath is organized in the same way as the one shown in Figure 2.1. However, because the height of the cells is fixed, it is possible that the number of available tracks, which are the horizontal wires used to route the connections over the cells, is less than the number of tracks required for routing. In this case, additional tracks must be added between the bit slices, and as a result, horizontal channels are inevitably inserted between bit slices. Hence bit slices are not guaranteed to abut upon each other in this system, as opposed to that in the general approach mentioned in Section 2.1.

The structure of a typical cell is shown in Figure 2.3. The design of the cell is flexible in that the pin positions are programmable. Pins are implemented as vertical strips in metal 1, shown as dash lines in Figure 2.3. A number of horizontal metal 2 tracks are placed over the cell. A pin is connected to a horizontal track by placing a metal 2-to-metal 1 via at the intersection of a strip and

a track. All of the tracks that a vertical strip can be connected to forms a set of *legal pin positions* for the corresponding pin, denoted by *LP(x)* for pin *x*. The tracks without vias on them can be used as feedthrough for over-cell routing.

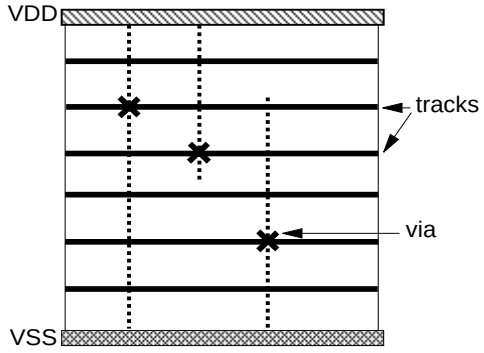


Figure 2.3 *The structure of a cell*

A linear placement algorithm, the A^* algorithm from the AI world, is used to place function blocks horizontally in order. The algorithm is in fact a branch and bound based search scheme. The search space is pruned by applying an inherent characteristic of the datapath — there is always a certain data flow from the input on the left to the output on the right.

The routing is carried out in a hierarchical partitioning fashion. Figure 2.4(a) shows a net that is routed in a bit slice. Each cell within the span of the net provides a horizontal track to the net, either as a real pin with a via or as a feedthrough without vias. The hierarchical partitioning scheme works as follows, as shown in Figure 2.4(b) and (c). First, a horizontal cut line is placed at the location such that about half of the available tracks are on each side of the cut line. The second step is called *terminal partitioning* in which the terminals (pins) on the net are partitioned into two parts and each of them is assigned to the corresponding subarea. This process is then repeated in each subarea until the subarea contains only one track. Figure 2.4(c) shows the sequence of how the routing in (a) is realized using this hierarchical strategy.

Terminal partitioning is essentially the central part of this approach. At each level of the hierarchy, the partitioning and assignment are carried through one net at a time. Pins are assigned to either side of a cut line according to the following criteria in the order of decreasing importance. The goal is to maximize the abutment and minimize the total wire length so as to

obtain the final layout with minimum area.

1. The number of pins assigned to either side of the cut line may not exceed the number of tracks available in each subarea.
2. There must be at least a legal position in the subarea for the corresponding pin.
3. To minimize the crossings over the cut line, all of the pins on a net (or part of the net) should be placed on one side as much as possible.
4. If a partial net can be placed entirely on either side of the cut line, the side with a higher attractive force is preferred. The force comes from other parts of the net that are located in the areas above or below the current area.
5. Two parts should have a good balance in the number of pins.

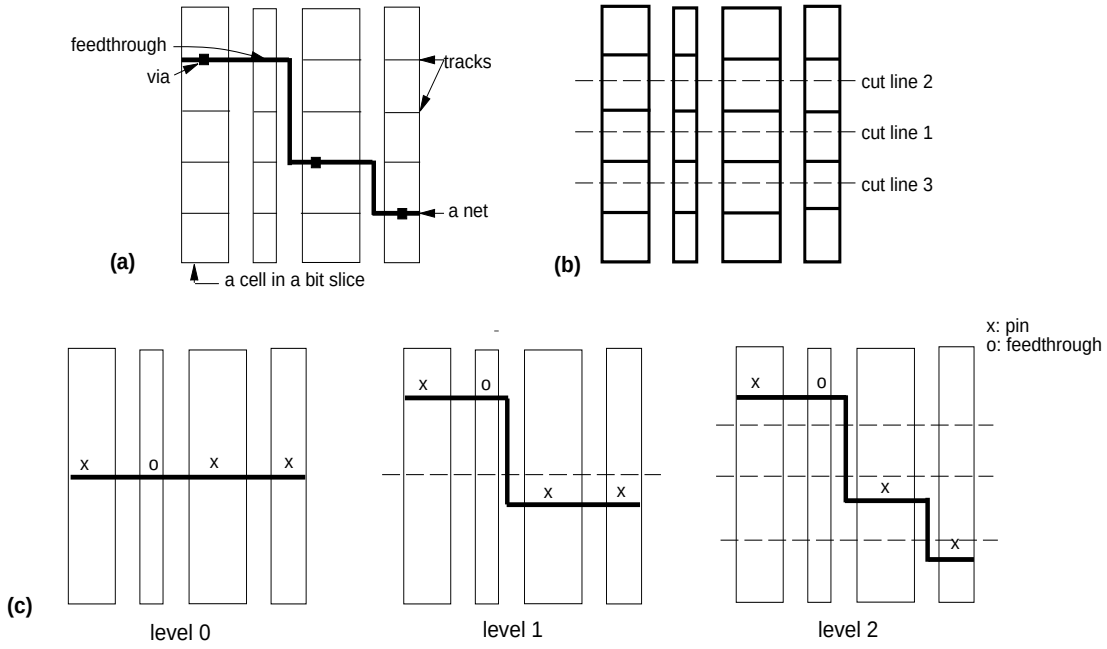


Figure 2.4 *An example of hierarchical partitioning and assignment*

During the process of partitioning and assignment at each level of the hierarchy, the placeability of the pins must be checked. It is resolved by finding the perfect matching for a bipartite graph $G(V,E)$ with $V = X \cup Y$ and $X \cap Y = \varnothing$, where X represents the set of real pins and Y the set of tracks. There is an edge between $x \in X$ and $y \in Y$ if $y \in LP(x)$ (see above for definition). A

maximum network flow algorithm is used to obtain the perfect matching.

*2.2.2 **Pathway** System*

This system places the functional blocks vertically, as a result, bit slices are formed in the vertical direction, as opposed to the arrangement shown in Figure 2.1. However, the concept of a bit-sliced structure is still unchanged. Figure 2.5 shows the partial layout of a datapath generated by *Pathway.* The datapath layout is generated through the following stages: placement, global routing, track assignment, and detailed routing.

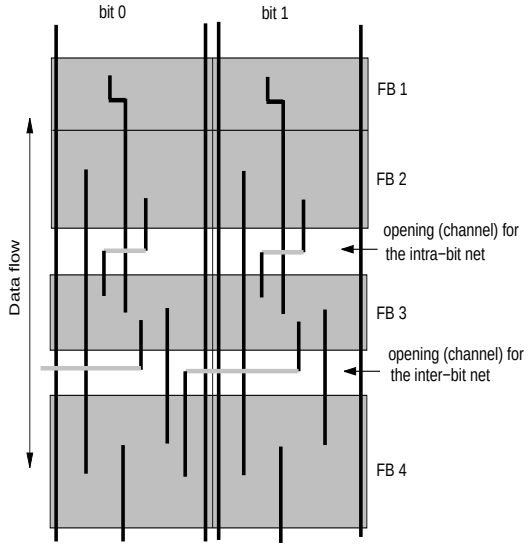


Figure 2.5 *A part of a datapath layout in **Pathway***

In *pathway,* the cells in the cell library are not so specifically designed as those in the previous system, as shown in Figure 2.6. Some pins may have several electrically equivalent options, whereas some may have only one. Pins are aligned to the vertical routing grid so as to be easily accessed by vertical metal 2 wires running over the cells.

The placement of the function blocks is manually specified by users.

The router is composed of a global router, a track assignment tool, and a detailed router, among which the track assignment is the primary part. **The global router** works on one net at a time, sequentially assigning the net either to the areas over the cell or to the channels. The intra-bit net,

which stretches within a bit slice, is simply assigned over the cells; the inter-bit net, which expands over multiple bit slices, is assigned by using a simple heuristic to eliminate the openings (channels) as much as possible. The heuristic states that each inter-bit net could generate only one opening along its span.

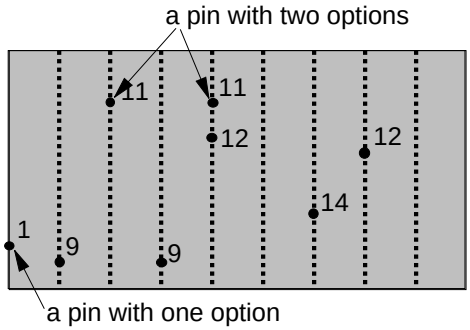


Figure 2.6 *The abstract view of a cell in **Pathway***

The Track assignment assigns nets to metal 2 tracks over the cells. The goal is to minimize the number of openings. A dynamic programming algorithm is used to accomplish this task, as described below. For a specific cell, a set of assignments of nets to tracks is defined as a *cell assignment*. The track assignment for a datapath can be seen as a process of choosing a *cell assignment* for each cell. The algorithm works by scanning cell by cell from the top of the datapath. For each cell, a table of all legal *cell assignments* is created, and the cost of each *cell assignment* is calculated based on a linear combination of total routing density and total net length over the cell. For a *cell assignment* q, the calculation is given by: $cost(q) = \underset{p \in P}{\text{Min}} \{cost(p) + cost(p \rightarrow q)\}$, where P is the set of legal *cell assignments* of the previous cell, *cost(p → q)* is the cost of routing from p to q. Accordingly, the cost of the top cell is zero, and the cost of the cheapest *cell assignment* of the bottom cell is the final result — the cost of the optimal track assignment of the entire datapath. When creating the legal *cell assignments* tables, a heuristic is used to limit the number of *cell assignments* that are actually considered for some “busy” cells. It chooses the net with no ports on the cell and assigns it to the track where it has only one option in all cells along its span.

A detailed router is used to route the horizontal channels and the areas over the cells. A channel

router is implemented by applying a contour-based variable width gridless channel routing algorithm, and the routing over each cell is solved as a maze routing problem with an optimization technique to move some connections from the channels to cells so as to shrink the channels further.

2.2.3 A Datapath Compiler Mixing Random Logic with Optimized Blocks

This system arranges a datapath in the same way as the general approach shown in Figure 2.1. Data buses are routed horizontally over the cells, and control signals, whose physical implementations are embedded within the layout of the cells, are routed vertically. This datapath compiler features the ability of mixing random logic with optimized blocks in the bit-sliced structure, and the flexibility of process technology independence. The latter is achieved by employing a symbolic layout approach backed up by a symbolic-to-real translation tool that maps the symbolic layout to a specific process automatically. The system takes the structural descriptions of the datapath as the input, passing through the placement and routing stages, and generates final layouts as the output.

The components of a datapath are either optimized blocks or bit cells. The optimized blocks, such as faster adders, register files, and barrel shifters, are obtained by calling the corresponding module generators with desired parameters. Both types of components provide virtual terminals on metal 2 tracks. Figure 2.7 illustrates the concept of the virtual terminal using an example bit cell: an inverter. In concept, a virtual terminal is the same as the programmable pin of the first system introduced in Section 2.2.1, which means a logical pin may correspond to several electrically equivalent access points located on different predefined tracks. In physical implementation, they have differences in that the virtual terminals of a logical pin are not necessarily positioned on the same vertical strip, and each terminal has three access types: left, right or both sides, whereas the programmable pins of a logical pin can only be located on one vertical strip without any access restriction.

The placement is performed in two separate steps. First, in each column, the relative locations of cells are specified manually with the goal to minimize the vertical channel areas; second, the

columns (function blocks) are placed using a simple pair-wise interchange algorithm. In this algorithm, a cost function (*CF*) is introduced: $CF = not(o) \times WL + o \times TO$, where *WL* is the total wire length, *TO* is the track overflow that is the difference between the number of the available tracks and that of the tracks required, o is a boolean variable that indicates if a track overflow occurs. Two columns are interchanged whenever *CF* is minimized.

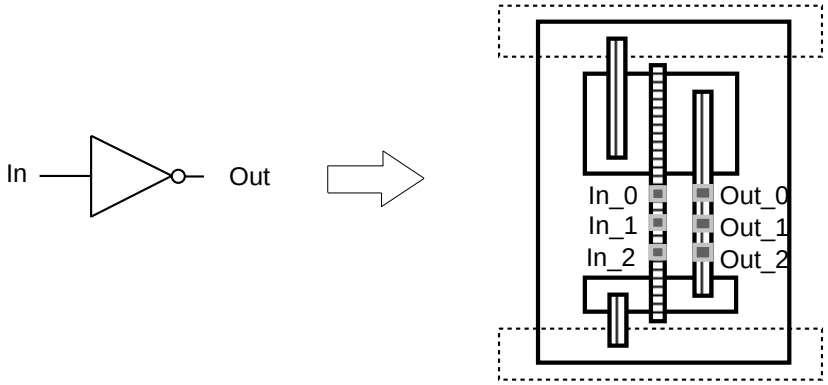


Figure 2.7 *The cell structure of an inverter with virtual terminals*

The router is composed of a global router implementing track assignment, and a channel router. The global router works on a symbolic grid within which some of the nodes are occupied by virtual terminals. First, it extracts the access type for each virtual terminal. Second, it scans the symbolic grid three times from left to right to route the nets over the cells. Each n-terminal net is divided into n-1 two-terminal nets. The first scan routes all of the opposite connections within each bit slice; the second scan connects the multi-track nets that are still in one bit slice but spans multiple tracks; the third scan routes the inter-bit nets that span multiple bit slices. Finally, a channel router is used to route all of the vertical connections. In the paper, the routing strategies in the three scans and the channel router were not discussed in detail.

2.2.4 A comparison of the three datapath compilers

All of the three datapath compilers generate the final layouts by passing through the placement and routing phases.

The first system provides the most complicated algorithm for placement; In the second system,

placement is specified manually; the third system implements the placement in a semi-automatic fashion.

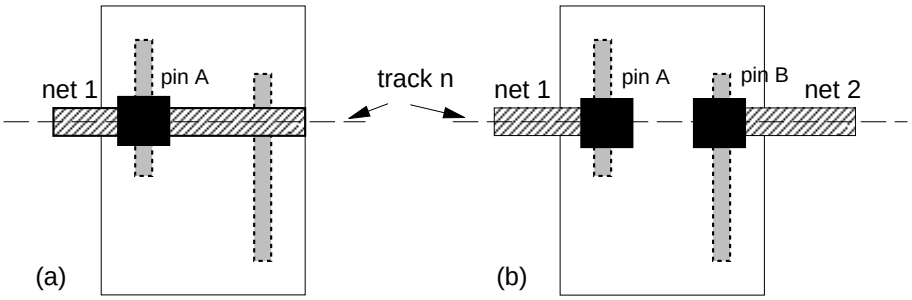


Figure 2.8 *(a) The track use in the first system's routing (b) A better use of track*

For routing, the first system adopts a hierarchical partitioning scheme. It is built on a theoretical basis, but has a drawback in the track use of the cells, as shown in Figure 2.8. Tracks, as precious routing resources, are wasted in this scheme. In Figure 2.8, there are two programmable pins on track n. Net 1 connected to pin A comes from the left; net 2 connected to pin B comes from the right. An efficient routing algorithm should be able to route both net 1 and net 2 if possible, as illustrated in Figure 2.8(b). However, the algorithm of this system can only assign the whole track of a cell to one net at a time, as shown in Figure 2.8(a). In the second system, the cell design is relatively simple. Each logical pin has only a small number of physical options. This simplicity results in a complicated routing process: global routing, track assignment, channel routing and maze routing over each cell. The track assignment uses a dynamic programming approach that not only scans cells one by one but also enumerates each cell's *cell assignments* along the scanning. Even though a heuristic is introduced to reduce the number of cell assignments for "busy" cells, it still has a major weak point in requiring large amounts of runtime. In the third system, although the routing strategy was presented without details, we do know that it is based on a symbolic grid that stores the information for virtual terminals. The algorithm scans the symbolic grid three times to complete the connections for all kinds of nets. Like all other symbolic grid-based algorithms, it suffers from large memory requirements. Scanning three times costs much runtime as well. Despite these disadvantages, this system did give us an inspiration to approach the routing problem for datapaths starting with the virtual terminals instead of nets to make best use of flexible cell design.

2.3 Routing Algorithms

As mentioned earlier, we have developed a new type of over-cell routing algorithm for datapaths in this thesis. The routing algorithm that makes best use of the specific cell design is the key to achieving a dense datapath layout. In the previous section, when introducing prior work on datapath compilers, we have used much space to describe their special routing strategies. In this section, we provide some necessary information about channel routing and over-cell routing. Section 2.3.1 introduces a dogleg channel routing algorithm with related channel routing concepts. Among many routing algorithms for different purposes, the channel routing algorithm, more specifically a dogleg channel routing algorithm, is discussed because we employ it in our system as a complement to the over-cell routing. Moreover, the idea of assigning connections to the minimum number of tracks in channel routing is an inspiration for us to develop our over-cell routing algorithm. Since over-cell routing is the main topic in the routing for datapaths, for reference and comparison, Section 2.3.2 gives a brief overview on general-purpose over-cell routing problems.

2.3.1 Channel Routing [Pre88]

Channel routing is generally classified as a type of restricted detailed routing. A channel is typically a rectangular space between two parallel rows of pins whose locations are fixed, as shown in Figure 2.9. Most channel routing algorithms assume that there are two layers available for routing. One of them is used for vertical wires, and the other for horizontal wires. The goal is to route all of the connections, while minimizing the height of the horizontally oriented channel, or the width of the vertically oriented channel. Before describing the algorithm, we list some terms in channel routing as follows, which are also illustrated in Figure 2.9.

- *Trunk* — A horizontal wire.
- *Track* — The line along which the trunks are placed.
- *Branch* — A vertical wire connecting the trunk to the top or bottom of the channel.

- *Dogleg* — A vertical wire connecting two trunks.

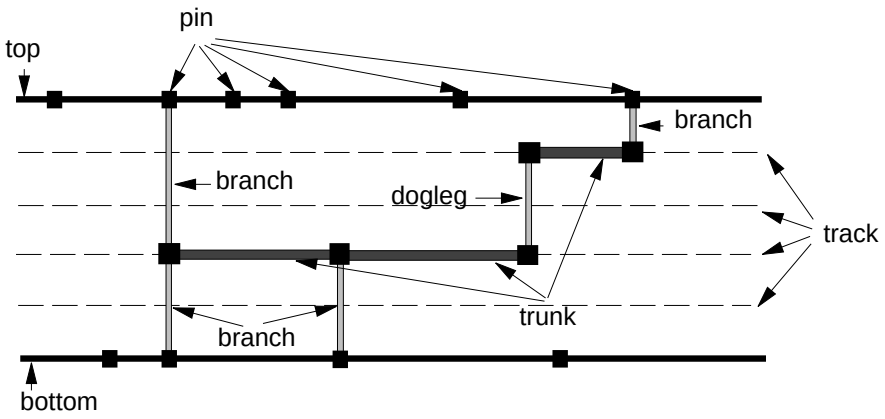


Figure 2.9 *A channel with one net routed*

Although there exists many channel routing algorithms, a dogleg channel routing algorithm is introduced in this section as an example because it is implemented in our system to complete the connections in vertical channels.

As mentioned earlier, the goal of the channel routing is to route all of the connections using as few tracks as possible. Earlier algorithms restricted each net to a single trunk section. However, this restriction may result in a solution with more tracks than necessary, as shown in Figure 2.10(a). Even though it is a simple example, it does indicate that this situation occurs because of the branch layer conflicts, which are generally called vertical constraints. In Figure 2.10(a), the trunk assignment should allow trunk C to be placed at the location indicated by the arrows without violating the horizontal constraints that prevent the trunks from overlapping. But, when adding the branch layers, the right end of trunk C needs a branch to connect to the bottom, while the right end of trunk B needs a branch to connect to the top. These two branches inevitably overlap. In fact, the vertical constraints must be satisfied because the pins are always fixed on a predefined grid before the routing begins. The vertical constraints of a channel are usually represented in a directed graph as depicted beside the channel. To solve the problem mentioned above, doglegs are introduced to break some nets into a series of subnets so that the number of tracks required for routing is reduced, as shown in Figure 2.10(b).

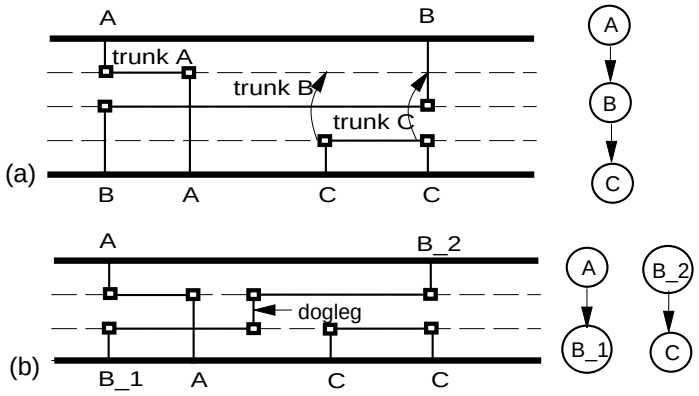


Figure 2.10 *Using doglegs to break constraint chains (a)without doglegs (b)with a dogleg*

However, finding the optimal number and location of doglegs has been shown to be NP-complete. A dogleg channel routing algorithm [Deu76] was proposed based on the following observation: the long constraint chains are usually caused by some crucial nets, such as clock signals, that are heavily connected to both sides of the channel. Thus, the algorithm decides to insert doglegs at only the terminal positions for these multi-terminal nets. This decision restricts the number of doglegs within a reasonable limit. Furthermore, the dogleg at a terminal position introduces one extra via as opposed to two extra vias at the other position, as shown in Figure 2.11. Based on this observation and decision, the algorithm starts by dividing all of the multi-terminal nets into a series of two-pin subnets. To limit the number of doglegs further, a parameter called *range* is introduced, which represents the minimum number of consecutive subnets that must be assigned to the current track.

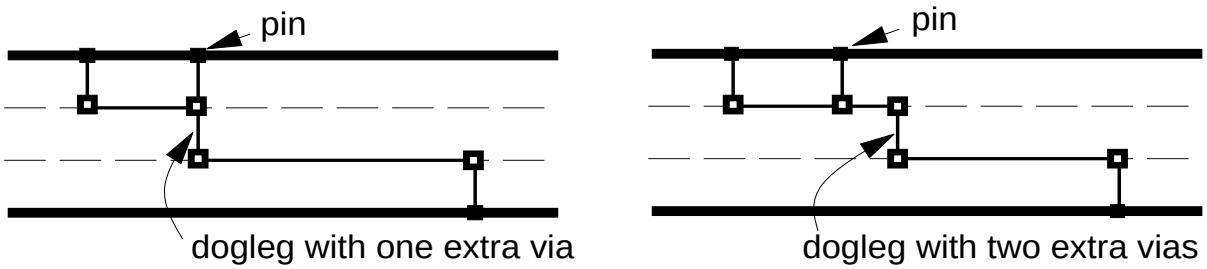


Figure 2.11 *Doglegs at different positions (a) terminal position (b) other position*

The dogleg channel routing algorithm could be described as follows, for the case when routing

from the top-left:

Partition the multi-terminal nets into a set of subnets according to the specified *range*.
Sort all of the subnets, or trunk segments, in increasing order of their left-most endpoints.
While the set of trunk segments is not empty
    Add a new track
    While scanning the sorted segments, the end of a track is not reached
        Check the unplaced segments encountered,
        If it is not constrained by some other as yet unplaced segments
            Place it to fit into the current track
    Endwhile
EndWhile
Add branches and doglegs

We implement this algorithm in our datapath compiler for the vertical channel routing. The *range* can be specified by the user.

2.3.2 General Over-cell Routing Problem

The general-purpose area routing approaches, such as maze routing, can be used for over-cell and over-block routing. However, for cell-based design in particular, most of the over-cell routing algorithms proposed are based on the channel approach [She95, Con90]. They rely on existing global routers to balance the usage of routing channels first, and then employ the over-cell routing in the detailed routing phase. Figure 2.12 shows an example channel with or without over-cell routing.

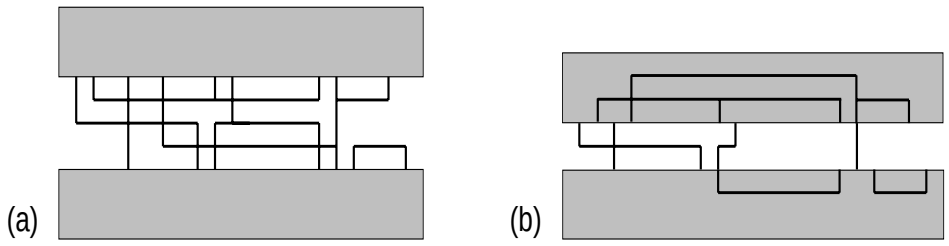


Figure 2.12 *An example channel: (a) without over-cell routing (b) with over-cell routing*

In the above-stated approach, there are four types of cell models, including *boundary terminal models* (BTM), *center terminal models* (CTM), *middle terminal models* (MTM), and *target-based cell models* (TBC), as shown in Figure 2.13.

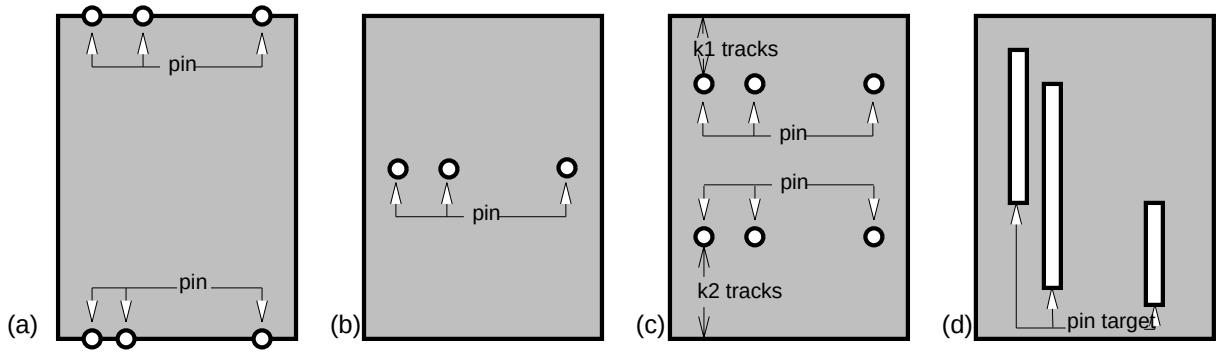


Figure 2.13 *Cell models (a) BTM (b) CTM (c) MTM (d) TBC*

The three models differ from each other in terms of terminal (pin) locations. In BTM, pins are at the cells' top and bottom boundary; in CTM, pins are located in the center of the cells, aligned horizontally; in MTM, pins are arranged in two rows, one of which is located k1 tracks below the upper cell boundary and the other is located k2 tracks above the lower cell boundary, aligned horizontally; in TBC, pins are in the form of vertical segments in the metal 1 layer. No matter which model is used, the goal of the over-cell routing is the same: to minimize or eliminate channel areas by maximizing the wiring over the cells. For BTM, CTM, and MTM, there exists algorithms for two, as well as three-metal-layer process technologies; for TBC, only algorithms for three metal layers have been proposed. The algorithms will not be detailed because we did not borrow any techniques from this general approach. The interested reader could refer to [She95].

2.4 A Cell Generation Tool: CLASSIC-SC

A commercial cell generation tool, CLASSIC-SC, from Cadabra Design Libraries Inc. is used to provide the cells needed in our datapath compiler [Cad99]. CLASSIC-SC is designed to generate cell layouts for standard cell libraries, while in our system we use it to generate cells on the fly.

CLASSIC-SC takes transistor-level netlists in SPICE or AL (the embedded control language of

CLASSIC-SC) format as the input, and outputs cell layouts in GDSII. The design flow using CLASSIC-SC consists of hierarchy extraction, placement, routing, and compaction, as shown in Figure 2.14. The tool is equipped with interactive editing tools to place the design process in full control of the users. Despite its flexibility, efficiency, and handcrafted quality in cell generation, there are two features in CLASSICSC-SC we would like to mention.

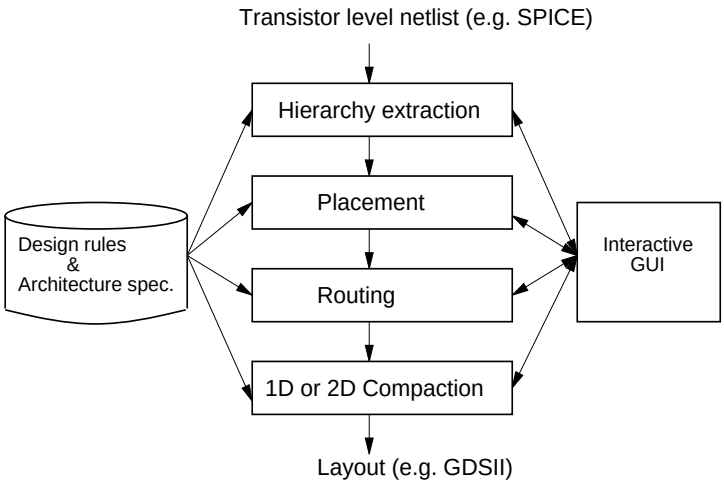


Figure 2.14 *Design flow using CLASSIC-SC*

One is its ability of easily retargeting to new process technologies, i.e., technology portability. Switching to a new technology could be as easy as writing a new set of technology and architecture specification files. A new technology corresponds to a new suite of design rules in Figure 2.14. This feature is achieved in CLASSIC-SC by using symbolic layout for placement and routing, combined with compaction as the last step to produce the final layout while enforcing the design rules.

The other is its *Application Language Interface*, which is delivered in the form of the AL embedded language. In CLASSIC-SC, most of the configuration information, such as netlists, technology descriptions, and cell architecture specifications, is specified in AL. Hence, it serves as a flexible interface for us to customize the tool's behavior at the request of some specific needs. AL also allows us to generate the cells in batch mode so that we can easily integrate this cell generation tool into our system.

2.5 Summary

In this chapter, previous work on datapath compilers was reviewed and related background information was provided. We conclude the highlights of the chapter as follows.

In Section 2.2, three major published datapath compilers are studied with respect to their overall structures, cell designs, placement and routing algorithms. Only the third system addresses the irregularity that exists in datapaths and the process technology independence issue. The cell designs in the first and third system are much more flexible with programmable pins and virtual terminals, respectively. This type of cell design provides a large configuration space for the router to explore. The routing strategy in the third system provided the inspiration for us to approach the problem starting with the virtual terminals.

Section 2.3.2 is only a glimpse at the over-cell routing problem in general. The routing in datapaths is different from that in general design because datapaths possess distinctive features for us to make use of. Moreover, the over-cell routing problem is solved on the basis of global routing, while for a datapath, we start from the beginning.

CHAPTER 3

A Datapath Compiler

This chapter presents the datapath compiler we have developed. We introduce the whole system in terms of its design capture, cell design, placement, routing, and important features. Section 3.1 describes the formulation of the problem. Section 3.2 presents the design capture style with a set of Java APIs based on an objected-oriented framework. Section 3.3 discusses the cell design using CLASSIC-SC. Section 3.4 introduces the placement directives primarily used to place random logic. Section 3.5 discusses the routing strategy we employed. A new type of over-cell routing algorithm for datapaths is proposed in this section. Section 3.6 summarizes the important features of the datapath compiler. Finally, Section 3.7 gives the summary of this chapter.

3.1 Formulation of the Problem

The main task of this thesis is to develop a datapath compiler based on the concept of module generation with special attention on flexibility. Like all other module generators, the datapath compiler produces layout geometric descriptions from input specifications. In our system, the input is captured at the structural level in the format of Java classes, and the output is generated as the layout geometric information in GDSII format. The flexibility is addressed in various aspects: parameterization, technology portability, the ability to deal with irregular datapaths with a small amount of random logic, and especially the future extension for timing optimization.

The datapath compiler arranges datapaths in the same way as that described in the general approach in Section 2.1 — functional blocks are placed horizontally; data are arranged to flow

from the left to right; control signals are introduced in the vertical direction. Layouts are produced by passing through the placement and routing phases. In our system, placement is specified manually in design descriptions. Adopting manual placement is based on the following considerations. First, the data flow in datapaths can always provide obvious indications for users to place functional blocks in one dimension. Second, specifying placement in design descriptions gives users a way to control the placement of random logic in datapaths, which is usually a small amount but can result in a complex problem for automatic placement. Figure 3.1 shows the composition structure of the datapath compiler. The shaded part has not been addressed directly in this thesis, but the whole system is designed for the future addition of this function without affecting the finished part.

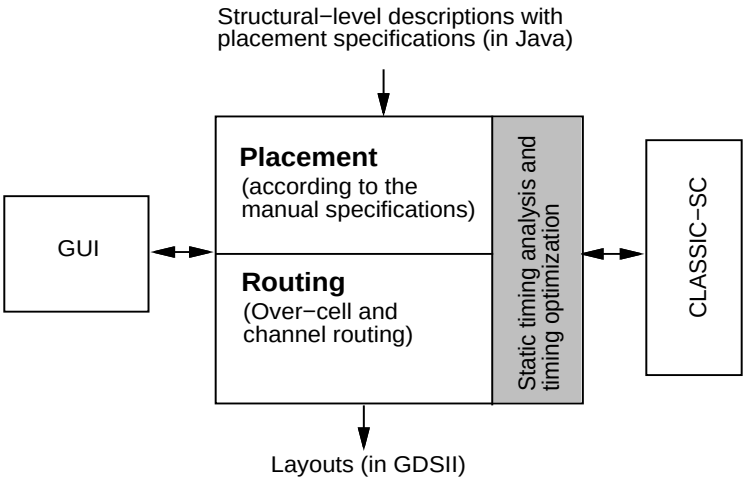


Figure 3.1 *The composition structure of the datapath compiler*

Despite the fact of being built upon the general approach, the datapath compiler does have one exception, which is caused by the way we design the components: the cells. As described in Chapter 2, traditional datapath compilers get cells from libraries that are specifically designed for datapaths. The physical implementations of control signals, which generally run in the vertical direction, are embedded within the layouts of the cells so that the control signals can be routed using only abutments. However, in our system, the cells produced by CLASSIC-SC do not have this characteristic because CLASSIC-SC is primarily designed for standard cell libraries. We choose CLASSIC-SC mainly because it is able to generate cells on demand, and as we will show

in Section 3.3, it allows the router to interact with it throughout the process of cell generation so that cells are produced according to the requirements of the router. Thus, in our approach, channels are needed to accommodate the routing of control signals. This scheme might result in larger area of final layouts, but it should be noted that the layouts of the cells with embedded control signals are certainly wider than that of ordinary cells of the same heights. As a result, we trade a little area for the flexibility that CLASSIC-SC provides and expect to make few sacrifices in area.

The datapath compiler is developed based on the TSMC 0.35-micron CMOS technology offered by the Canadian Microelectronics Corporation (CMC). It seems that using what technology should not be an issue because the datapath compiler is claimed to be technology portable. However, technology is an issue in terms of how many routing metal layers it provides. Generally, for technologies with different numbers of metal layers, different types of routing strategies are needed to generate dense layouts by making the best use of metal layers. Although it is possible to develop a datapath compiler that is technology portable not only in the sense of obeying the design rules but also in the sense of adopting the most suitable routing algorithm for the technology, we expect to accomplish this in the future and we did take this into account when doing the object-oriented design of the system so that it allows the future addition of other routing algorithms. The routing algorithm in this thesis is developed for the technologies such as TSMC 0.35-micron technology with three metal layers. For routing, three metal layers are allocated as follows. The metal 3 layer is saved for block-level routing because datapaths always serve as functional blocks in large designs and we do not want to make their layouts large blockages at the block level. The metal 1 and metal 2 layers are used to complete the layouts of datapaths: the metal 1 layer is used for cells, metal 2 for over-cell routing, and metal 1 together with metal 2 for channel routing. For technology that provides more metal layers, new routing algorithms might be developed to ultimately eliminate channels and complete the overall wiring by routing over the cells, and hence achieve final layouts using only abutments.

The Java language is chosen to implement the datapath compiler. We chose Java mainly because it is a pure object-oriented language, which fulfils our goal of object-oriented design. Other reasons

include its portability, and its direct support for the development of Graphical User Interfaces (GUI).

3.2 Design Capture

A set of Java classes has been developed for design capture. It, in fact, forms the object-oriented framework that serves as the foundation for the whole system. The implementation details will not be discussed in this thesis, while the classes and their relationships will be described in the next Chapter. In this section, when discussing the classes, we focus on the aspects that are significant to users and show how to use them in capturing datapath designs.

The goal of developing such a set of Java classes is to provide a set of Java APIs that can be used to capture datapath designs at the structural level with placement specifications, and to aid the work in the following routing phase. The classes are carefully designed such that datapaths can be captured concisely, and the users with little or even without Java knowledge are able to use them without difficulties. Example 1 shows how a type of parameterized register is captured in the datapath compiler. The register uses a D flip-flop class called dff, which describes a type of basic cell.

Example 1: *The Java class of a parameterized register*

```
class Register extends FuncBlock {
    Register(String name, int bits, Net in, Net out, Net clk) {
        setName(name);
        for(int i=0; i<bits; i++) {
            addSubModule(new dff(in.ripNet(i), clk, out.ripNet(i)));
        }
    }
}
```

Note that cells are the basic constituent elements of a datapath, and hence in addition to their class entities, their transistor-level netlists are also needed for generating the layouts. The class entity of

a cell is simpler than that of a functional block or datapath, as we will show in the next subsection.

In the previous example and all the following examples of this Chapter, the names of the classes, which are from the framework we created, are shown in a bold fond, and the members of these classes, including attributes and operations, are shown in an italic font.

In general, a datapath consists of a set of functional blocks, and each functional block consists of a set of cells. A functional block can be treated as the special case of a datapath, i.e., a datapath consisting of only one functional block. Thus, only the cell and datapath are captured differently, as we will discuss in the following two subsections.

3.2.1 Capturing Cells

To capture a type of cell, two pieces of information are needed. One is its inner structure expressed as a transistor-level netlist in SPICE or AL format; the other is its outer interface depicted as an entity in Java class format. In general, the class written for a type of cell contains two constructors, whose principal function is to specify the directions of its IO ports and the drive strength of the cell. Example 2 shows the Java class of a 2-input AND gate.

Example 2: *The Java class of a 2-input AND gate*

```
class and2 extends LeafCell {
    and2(Net a, Net b, Net out) {
        addPort("a", Port.IN, a);
        addPort("b", Port.IN, b);
        addPort("out", Port.OUT, out);
    }

    and2(int driveFactor, Net a, Net b, Net out) {
        this(a, b, out);
        setDriveFactor(driveFactor);
    }
}
```

From this class, users can instantiate as many AND gates as they want in a design. For instance, a statement of "new and2(n1, n2, n3);" creates a 2-input AND gate of size 1, which refers to the lowest drive strength, where n1, n2, and n3 are the nets that ports "a", "b", and "out" are connected to, respectively. To obtain the same gate of size 2 with higher drive strength, simply use "new and2(2, n1, n2, n3);".

Through this example, we also show how straightforward it will be to add the timing optimization function into the existing system because the cell sizing process could simply be accomplished by instantiating the corresponding cell based on the result of static timing analysis and optimization. Instantiating various cells of different sizes is a dynamic process when the program is running. Note that for a type of cell with various drive strengths, we only need one class, while we need various netlists corresponding to different drive strength requirements. Since these netlists are generally the same in topology but different in transistor sizes, it is easy to let the system automatically generate the netlists for higher drive-strength cells from that of a size 1 cell when the timing optimization function is added into the system in the future.

In summary, a type of cell is completely defined by a class and a set of transistor-level netlists. A cell is obtained by instantiating an object from the class with the drive strength required.

3.2.2 Capturing datapaths

As mentioned earlier in this section, a functional block can be captured in the same way as a datapath. Example 1 illustrated how a typical functional block such as a register can be captured in the datapath compiler. Example 3 below shows how a datapath is captured in general.

A datapath design is generally captured in a class that consists of only one constructor, as shown in the example. The statements in the constructor can be grouped into four parts according to their functions. The first part is to set some characteristics, such as the name and the number of bits; the second part is to add IO ports, which is mainly used to indicate the types of the ports — *data* or *control*. This indication is necessary because the routing algorithm treats data and control nets differently when choosing routing directions. The third part is to create the nets used to

interconnect functional blocks and sub-datapaths. The last part is to add sub-modules, including functional blocks and sub-datapaths.

The capturing style is inherently hierarchical. As shown in Example 3, DataPath12 is a sub-datapath whose structure is described in another Java class. Note that only at the first level of the hierarchy is the second part (where the IO ports are added) needed.

Example 3: *The Java class of an example datapath*

```
class DataPath1 extends DataPath {
    DataPath1(String name, int bits, Net in, Net out, Net clk) {
        //Part 1: set characteristics
        setName(name);
        setBits(bits);
        //Part 2: add IO ports
        addIOPort(in, Port.DATA);
        addIOPort(out, Port.DATA);
        addIOPort(clk, Port.CONTROL);
        //Part 3: create new nets
        Net n1 = new Net("n1", bits);
        addNet(n1);
        //Part 4: add sub-modules
        addSubModule(new DataPath12("DataPath12", bits, in, n1, clk));
        addSubModule(new Register("Output", n1, out, clk));
    }
}
```

A datapath with a specific number of bits can be obtained by instantiating an object from this class using a statement like, for example, "new DataPath1("dp1", 8, n1, n2, clock);", where n1, n2, and clock are the input, output, and clock net connected to the datapath, respectively.

Placement information is also captured in design descriptions in our system. The capturing style of this part will be presented in Section 3.4 when the placement scheme is discussed.

3.3 Cell Design

This section describes how we use CLASSIC-SC to generate cell layouts for over-cell routing. As mentioned earlier, over-cell routing is the key to achieving highly dense layouts by taking advantage of the regularity in datapaths, while cells are always required to be specifically designed for this purpose. The idea behind this so-called specific design is always the same, i.e., the design should enable a cell to provide several electrically equivalent access points for a logical pin on different predefined tracks. The programmable pins and virtual terminals of previous datapath compilers introduced in Chapter 2 are based on this idea as well. In this thesis, by using CLASSIC-SC, we achieve this goal in a different way.

Generally, CLASSIC-SC produces a cell by passing through hierarchy extraction, placement, routing, and compaction. The whole process can be completed in batch mode by submitting an AL script. In our approach, to allow the router to interact with CLASSIC-SC, we reschedule the process of designing a cell as follows.

1. **An AL script is executed to generate a cell without IO ports by passing through hierarchy extraction, placement, routing, and compaction, and then the cell is exported in GDSII format**. In the ordinary cell design using CLASSIC-SC, ports are added manually or automatically in the routing stage, but here we deliberately exclude this step from the process. In CLASSIC-SC, the stages before compaction work on the symbolic layout where design rules are not enforced. In the compaction stage, the compactor moves components and wires in the plane to ensure that both the design rules are obeyed and the compaction objectives, such as minimizing the width and wire length of a cell, are reached. However, CLASSIC-SC allows a compacted layout to be set as editable, which means we can still treat a compacted layout as a symbolic layout, hence we can add more components into it, and then compact it one more time. This feature is exploited in the following steps.
2. **The router of the datapath compiler reads in a cell's GDSII file, and then extracts all possible pin locations for each IO port of the cell.** The possible pins are located at the intersections of predefined horizontal tracks, which is determined by the target

technology, and the vertical poly or metal 1 layers that carry the signals of the corresponding ports, as shown in Figure 3.2(a). The legal pins marked in the figure are a subset of the possible pins.

3. **By running another AL script, CLASSIC-SC checks all the possible pins to see if each of them can be added into the cell's layout**. Because this step still works in the symbolic layout, design rules are not used as the criteria for checking, instead, as long as the addition of a pin (a poly contact and a metal1-metal2 via, or a metal1-metal2 via) does not incur a short circuit in the layout, the pin will be accepted as a legal pin. Thus, the constraints are not so tight and we can get a fairly large number of legal pins as the result.
4. **Based on the information of available legal pins, the router carries out its algorithm, and finally passes its decision of final pin locations for each port back to CLASSIC-SC**. The over-cell routing algorithm will be discussed in detail in Section 3.5.
5. **By executing another AL script, CLASSIC-SC places the pins for the locations specified by router, compacts each cell again, and finally generates the layout for each cell by finishing clean-up steps**. Note that during the compaction process of this step, CLASSIC-SC has been configured such that pins can only move horizontally. This restriction comes from the router, which strictly requires the pins stay on the horizontal tracks specified in Step 4. In CLASSIC-SC, it is not unusual for the compaction step to get an infeasible result simply because design rules cannot be met. However, in our approach, because this second compaction is based on an already compacted layout and hence the regions which need to be adjusted are only around the locations where the pins are added, the probability of getting an infeasible layout is low. If an infeasible layout does come out at this stage, currently we can only get around this by stretching cells, while we think we should be able to prevent this situation from occurring if CLASSIC-SC gives direct support at a lower level for this kind of use.

Figure 3.2(a) shows the layout of an AND gate without ports after the first compaction. Figure 3.2(b) shows the layout with ports after the second compaction. Figure 3.2(c) shows the final layout after clean-up steps.

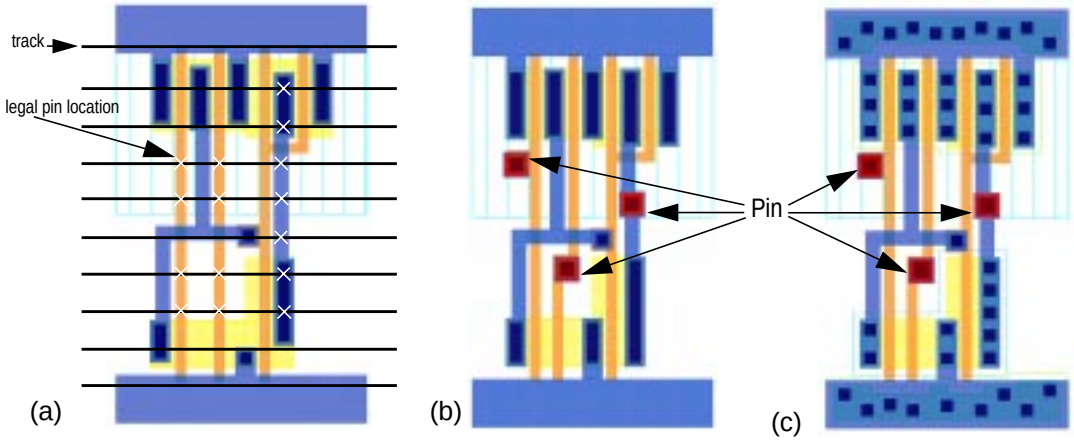


Figure 3.2 *The layout of an AND gate (a) after first compaction (b) after second compaction (c) final layout*

In summary, we complete the cell design in the process using collaboration between the router and CLASSIC-SC. Compared with traditional methods [Tsu94, Amm93, Cai90, Coh93], including those of the programmable pins and virtual terminals introduced in Chapter 2, our approach has advantages not only in the technology portability and the flexibility of generating a cell on demand (which largely benefits from using CLASSIC-SC) but also in the quality of the final layouts produced. More specifically, in traditional methods the programmable pins or virtual terminals are designed such that the later additions of pins to their locations will not cause any design rule violation in cells. This type of design inevitably results in larger layouts than necessary because only some of the virtual terminals will finally be used in practice. On the contrary, our approach gets decisions from the router and places the real pins in the symbolic layout where design rules are not yet enforced. Final layouts are obtained after another compaction. As a result, there are no sacrifices of area for the flexibility to be exploited in routing. The flexibility of CLASSIC-SC makes this approach possible.

3.4 Placement Scheme

In our datapath compiler, placement is specified manually in the design descriptions. Note that placement here means specifying relative locations rather than absolute coordinates for functional blocks and cells. Thus, the sizes of the components need not be known when specifying the

placement. The actual calculation of precise physical coordinates can be deferred until the routing phase is completed and the real sizes of cells are determined.

Because of the regularity of datapaths, the relative placement scheme is rather simple. In our approach, the placement of functional blocks is implicitly specified from the left to right by the instantiation sequence of the functional blocks in the design descriptions, i.e., the first functional block is placed at the leftmost position, the second is placed at the second location to the left, and so on. As well, in each functional block, where cells are arranged vertically, the placement of cells is implicitly specified from the top to bottom by the instantiation sequence of the cells. This placement scheme is consistent with the arrangement defined in Section 3.1.

As mentioned earlier, in our approach a small amount of random logic is allowed in the bit-sliced structure. Since the bit-sliced structure is still retained, irregularity just needs to be addressed at the cell level in each functional block. A mechanism is added to the placement scheme to deal with this situation. In a functional block, users can explicitly specify the placement strategy by invoking a method called *setPStrategy(p),* where parameter *p* is a variable of type *Placement,* which is an interface in the framework. Currently, we have implemented two classes called *RegColumn* and *CustomColumn* for interface *Placement*. For extension in the future, more placement strategies such as “place the cells using simulated annealing” may be added to the framework as long as their corresponding classes implement interface *Placement.*

RegColumn and *CustomColumn* together are sufficient for our current use. *RegColumn* is used for a placement that is relatively regular. For example, the cell placement shown in Figure 3.3(a) can be simply specified in the class of a functional block as follows:

```
Placement p = new RegColumn(1, 1, 1);
setPStrategy(p);
```

The first parameter of *RegColumn()* is the offset from the top, the second parameter is the gap between two neighboring cells, and the third parameter is the range of a cell, as illustrated in Figure 3.3(a). The unit of these parameters is the height of a standard cell in the current design, which might change during the routing process. *CustomColumn* is used for a placement that is

relatively irregular and cannot be described simply by using *RegColumn*, as shown in Figure 3.3(b). To capture this kind of placement, we generally partition it into several sub-templates, each of which can be described using two parameters: gap and range. Accordingly, the placement shown in Figure 3.3(b) can be specified in the class of a functional block as follows:

```
Placement p = new CustomColumn();
p.addSubTemplate(0, 2);
p.addSubTemplate(1, 4);
setPStrategy(p);
```

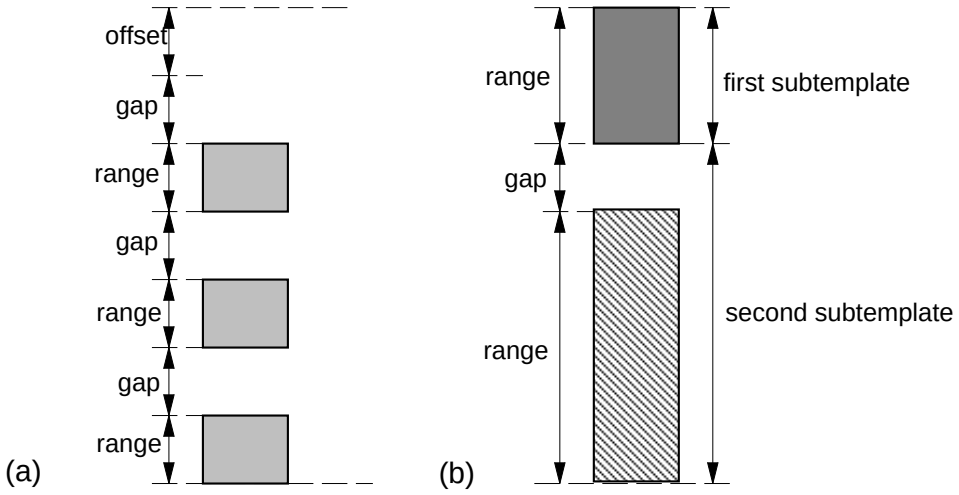


Figure 3.3 *Placement strategy (a) regular (b) irregular*

3.5 Routing Strategy

This section discusses the routing strategy we developed for the datapath compiler. Routing is to complete the electrical interconnections between the components that are arranged on a plane after the placement. Generally, in addition to making the routed result function, there are several routing aspects to be optimized, including the area, timing, and power consumption of the final layout. As mentioned earlier, the main issue addressed in this thesis is to generate dense layouts using over-cell routing for datapaths. The cell design in our approach has been described in Section 3.3, which is an interacting process between the router and the cell generation tool CLASSIC-SC. In this section, we describe how the router works and how it cooperates with

CLASSIC-SC to produce dense layouts.

The entire routing problem is resolved by employing a combination of over-cell and channel routing. Most of the interconnections in datapaths are completed during the over-cell routing process. Because the over-cell routing algorithm is mainly based on the interval graph, a summary of the necessary and sufficient graph theoretic fundamentals, especially algorithms for the interval graph, is given briefly before the routing algorithm is discussed.

3.5.1 Finding Maximum Independent Set for Interval Graphs

A graph is generally represented by $G = (V, E)$, where V is a finite set of $|V|$ elements called *vertices* and $E \subseteq \{\{x, y\} | x, y \in V, x \neq y\}$ is a set of $|E|$ unordered vertex pairs called *edges*. For distinct vertices x and y, x is said to be *adjacent* to y (or equivalently, y is said to be *adjacent* to x) if $\{x, y\} \in E$; otherwise, they are said to be *independent*. A set $V' \subseteq V$ of vertices is called an *independent set* if the vertices in V' are pairwise independent. A *Maximum Independent Set (MIS)* is the one with a maximum number of vertices among all independent sets.

The concept of an *interval graph* is defined as follows. A graph $G = (V, E)$ is called an *interval graph* for a family $S = \{S_i\}_{i=1}^{n}$ of intervals on the real line if there is a one-to-one correspondence between V and S such that two vertices are adjacent if and only if the corresponding intervals have a nonempty intersection. Accordingly, the method of finding the maximum number of non-overlapping intervals on a line is the same as that of finding an MIS for its corresponding interval graph. For general graphs, the problem of finding an MIS is known to be NP-hard; while for interval graphs, an algorithm of $O(n\log n)$ time complexity was proposed in [Gup82], which is described below.

For an interval graph with n intervals, the algorithm first sorts the 2n endpoints in ascending order of their values. It then scans this list from left to right until it first encounters a right endpoint. It then outputs the interval having this right endpoint as a member of a maximum independent set and deletes all intervals containing this point. This process is repeated until there is no interval left in the list.

As described below, we apply this algorithm to over-cell routing.

3.5.2 Overall Routing Strategy

The router in our system is composed of an over-cell router and a channel router. The latter serves as a complement to the over-cell routing. The routing algorithm is developed based on a three-metal-layer technology such as the TSMC 0.35-micon CMOS technology. The metal 3 layer is saved for the block level wiring, so we use metal 2 for over-cell routing, metal 1 and metal 2 for channel routing, and metal 1 and poly for cell design. To achieve highly dense layouts, we set the overall routing strategy as follows.

First, the over-cell routing algorithm should be able to make the best use of the cell design. The example in Figure 3.4(b) shows what result should be produced by the over-cell routing for an intra-bit net. Net 1 is an intra-bit net, and to avoid opening a channel it should not be routed as shown in Figure 3.4(a). However, none of the previous datapath compilers could completely avoid this kind of wiring in their over-cell routing algorithms. In our routing strategy, intra-bit nets will not incur openings (channels), as detailed in the next subsection. The routing for all of the intra-bit nets are completed horizontally over the cells. However, if the cells in a bit slice do not have sufficient legal pins for reaching this goal, they will be automatically stretched in the vertical direction by informing CLASSIC-SC to produce cells with a new height.

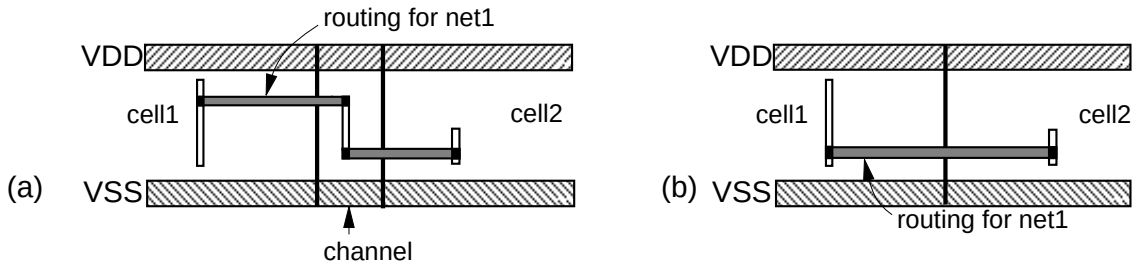


Figure 3.4 *Routing intra-bit nets(a)Unacceptable result with an opening (b)Acceptable result*

Second, channels are inevitably needed when routing inter-bit and control nets that run vertically. As stated earlier in Section 3.1, compared to the general approach there is one exception in our datapath compiler — the physical implementation of control nets are not embedded in the layouts

of cells, hence channels are needed to route these nets. To make the abutment as much as possible so as to make dense layouts, we establish a rule to introduce channels between functional blocks (FBs). As shown in Figure 3.5, the channels can only be introduced between the first and second FBs, the third and fourth FBs, and so on; or equivalently, between the second and third FBs, the fourth and fifth FBs, and so on. For instance, in the example shown in Figure 3.5, the control nets of the first FB are routed in the channel on its right, and that of the second FB are routed in the channel on its left, whereas there is no channel on the right side of the second FB. Channels are introduced only when necessary. However, the rule stated above may be violated when routing some inter-bit nets. Figure 3.5 shows two such inter-bit nets that can only be routed in an additional channel if we do not want to incur a timing penalty by routing them in a routine channel nearby.

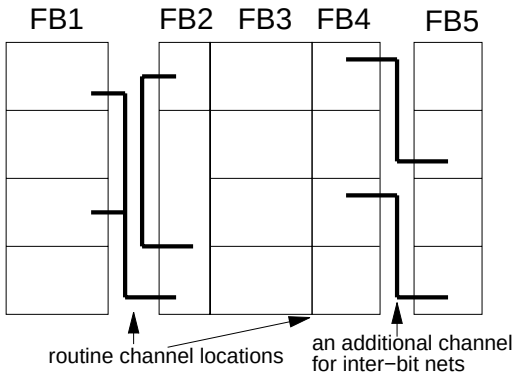


Figure 3.5 *The location of channels*

Third, unlike those of the previous datapath compilers, our routing strategy has to deal with both intra-bit nets and control nets during the over-cell routing phase. To route control nets in the vertical channels, horizontal wiring segments are needed to lead the pins, which are on the control nets in different cells, to the left or right edges of the cells. These horizontal routes in fact consume considerable routing resources over the cells. Thus to make the best use of routing resources, the over-cell routing algorithm has to deal with control nets together with intra-bit nets rather than route them sequentially.

In Figure 3.6, a simple example is used to illustrate the result of our routing strategy. Figure 3.6(a)

shows the abstract view of the example datapath. Note that the second and fourth functional blocks contain irregular cells. Figure 3.6(b), (c), (d), (e) are obtained directly from the datapath compiler. Figure 3.6(b) shows its final layout. All of the routes completed by our routing algorithm are shown in Figure 3.6(c). To make it clear, we show the over-cell routes and channel routes separately in (d) and (e).

In the following subsection, we will discuss the over-cell routing algorithm in detail.

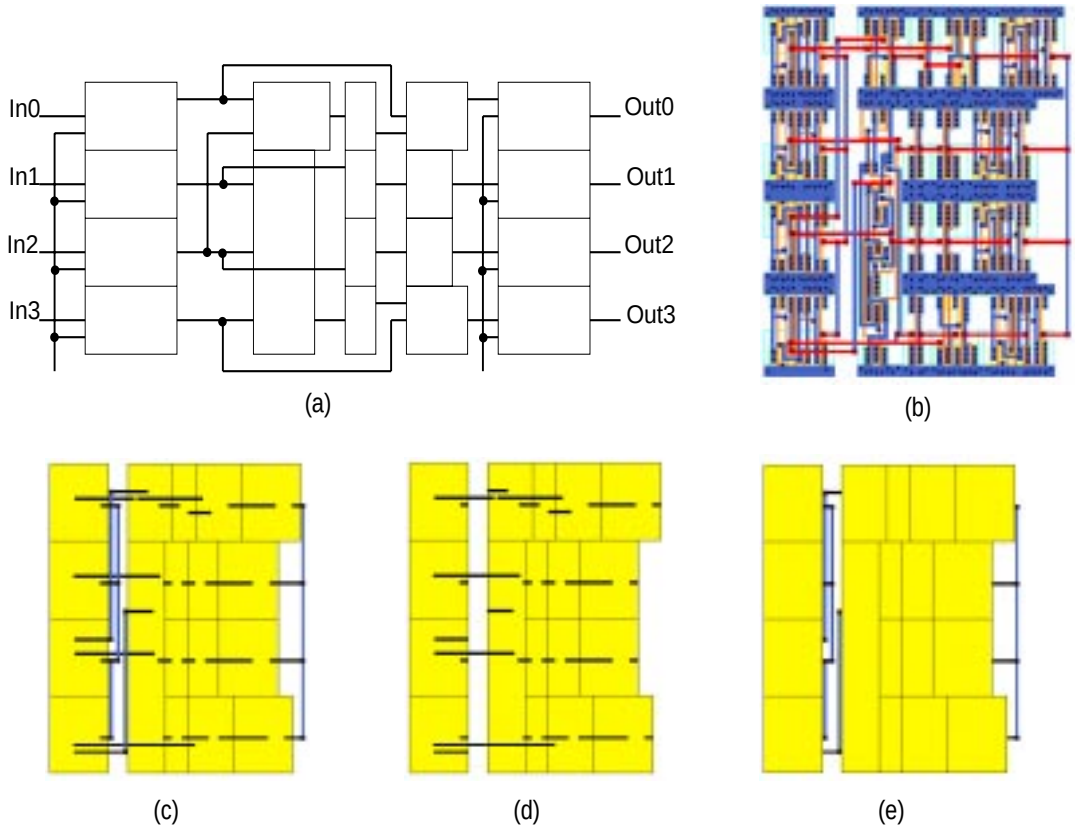


Figure 3.6 *Routing for an example datapath (a) The abstract view (b) Layout (c) All routes (d) Over-cell routes (e) Channel routes*

3.5.3 Over-cell Routing

The over-cell routing in our approach is to route all of the horizontal wires over cells so that these wires plus wires in channels complete all of the interconnections of a datapath. The entire over-cell routing process is composed of two phases. In the first phase, the algorithm works row by row (bit slice) to route intra-bit nets and parts of control nets (as explained in the last subsection) using

horizontal wiring segments over cells. If the routing in a row cannot be 100 percent completed, the cells in the row will be stretched automatically until it is successfully routed. The second phase is to route the over-cell horizontal wires for inter-bit nets. As well, stretching might be necessary to complete the interconnections.

In the first phase, the algorithm deals with one bit slice at a time. Figure 3.7 shows a typical bit slice with over-cell tracks depicted on it. A *track* is a horizontal line along which over-cell wires are placed. The space between two adjacent tracks is calculated as the via-to-via pitch of the technology being used, as shown in Figure 3.8.

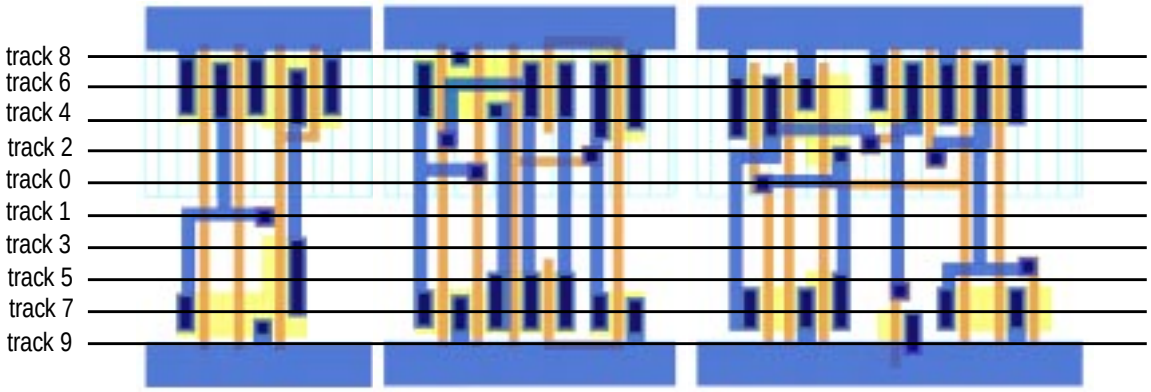


Figure 3.7 *A bit slice with tracks depicted on it*

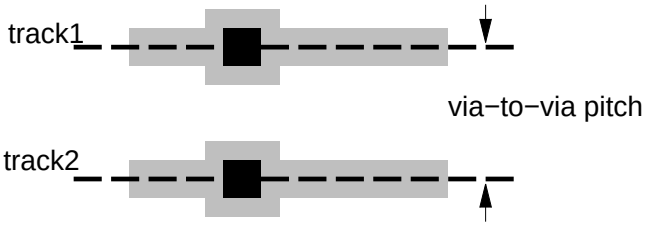


Figure 3.8 *Via-to-via pitch used to calculate the space between adjacent tracks*

In each row, the algorithm works on one track at a time starting from the state that is shown in Figure 3.7. In this state, cells in the row have been generated and the legal pins for each IO port of the cells have been found by passing the first three steps of the cell design procedure described in Section 3.3. The tracks are chosen in a symmetric manner starting from the middle of the bit slice. In Figure 3.7, the number of tracks indicates the choosing sequence, which commences from the

middle and then alternates between the top and bottom. Adopting this scheme is mainly because poly contacts in P and N active areas are forbidden in MOS process technologies (including the technology we use), hence the middle region between p-transistors and n-transistors possesses the most choices for IO ports.

The routing on each track is further divided into two phases. In the first phase, an interval graph is constructed for the net segments on the track, and then in the second phase, the algorithm described in Section 3.5.1 is used to find the MIS of the constructed interval graph.

From the state shown in Figure 3.7, the first phase starts by scanning the bit slice from the left to right, asking each cell to reply with its legal pins on the track. Each pin is in fact an object that contains all necessary information for later use, including the net it connects to, the cell it resides in, the name of the port, the status of being routed or unrouted, etc. These pins are stored in a list, and then the algorithm proceeds by scanning the list from the left to right until it first encounters an unrouted pin. If the unrouted pin is on a control net, a net segment (an interval that acts as a vertex of the interval graph being constructed) is obtained by spreading a horizontal wire from it to the edge of the cell, beside which there is a routine channel. Otherwise, another scan is performed backwards along the list from the unrouted pin to the left until either a pin, which is on the same net as the unrouted pin, is found or the left end of the list is reached. In the former case an interval, whose left and right endpoint are the end and beginning of the backward scan respectively, is identified as a vertex of the interval graph being constructed. The first scan (the forward scan) continues until it reaches the end of the track. The final result is the interval graph constructed for the track. Using the second backward scan is based on the following considerations.

First, rather than decompose an n-terminal net sequentially into a set of n-1 two-terminal nets before the routing algorithm begins, the algorithm completes this task dynamically so that the routing resources, which are the over-cell routing tracks in our case, can be used more efficiently to achieve 100 percent of the connections. Figure 3.9 shows an example. There are three terminals on net n1 connecting port A in cell 1, port B in cell 2, and port C in cell 3 together. Three cells are aligned horizontally in a bit slice. If we divide n1 sequentially into two two-terminal nets: n1_1,

and n1_2, as we can see, there is no way to connect port B of cell 2 with port C of cell 3. However, in our approach using the backward scan enables us to complete the connection finally as shown in Figure 3.9. Second, instead of scanning in both directions, we restrict the direction to the left because, as we have mentioned earlier, the functional blocks of a datapath are placed according to the data flow, in our case, from the left to right. Thus, scanning to the right might result in unexpected long paths. For instance, the routing result of a long path between A and B in Figure 3.10(a) is obtained by scanning to the right, while by scanning only to the left, the same connection is completed with a better result from the perspective of timing, as shown in Figure 3.10(b). Although this restriction might reduce routing choices to a limited extent, we do not want to make such sacrifices of performance for routing completion, which, on the other hand, can be achieved by stretching cells.

In the second phase, the algorithm proceeds to find the MIS of the interval graph by employing the algorithm introduced in Section 3.5.1. Whenever an interval is chosen to be a member of the MIS, the pin which corresponds to its right endpoint is set to be as routed. According to the definition of MIS, the result is in fact the maximum number of non-overlapping net segments we can put on the track.

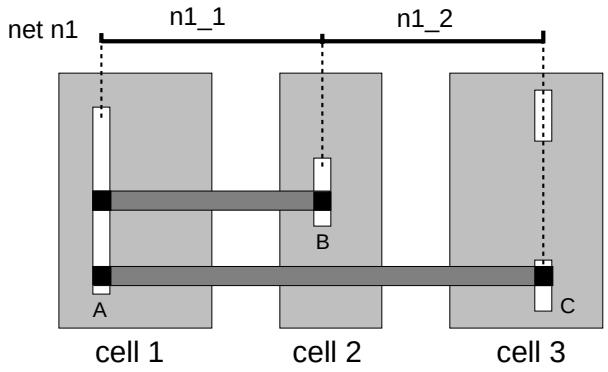


Figure 3.9 *A routing example by decomposing a net dynamically*

Once the routing on one track is finished, the cells in the bit slice are scanned one more time from the left to right to check their routing status. Each cell is requested to inform the router about the routing status of its IO ports that reside on intra-bit and control nets. Based on the answer from all cells, the router either moves on to the next bit slice or stays and works on a new track. This

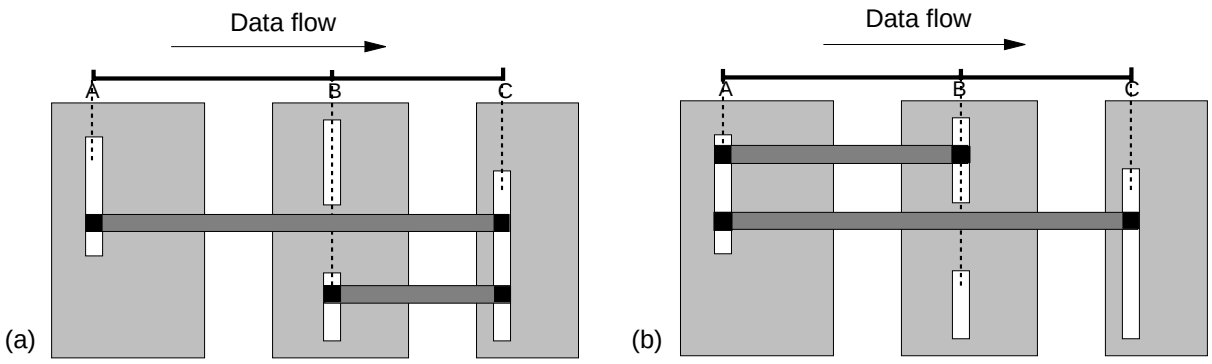


Figure 3.10 *(a) Long routing wire between A and B (b) A better solution*

process continues until all bit slices are routed. The routing result consists of two parts. The first part is the information of the port locations of the cells, which are kept in each cell and then used by CLASSIC-SC to generate the cell's final layout with ports (see steps 4 and 5 in Section 3.3). The second part is a set of wiring segments whose information is kept in the router, which is an object of type *Router.*

When implementing the algorithm, special attention has to be paid to deal with the tracks of flipped cells and especially big cells whose height is several times bigger than that of the standard cells. A mapping is needed to assign the logic tracks of a bit slice to the real tracks in a cell's layout.

After all of the intra-bit and control nets are routed for all bit slices, the router proceeds to complete the connections for inter-bit nets. An inter-bit net is a net or a partial net whose terminals reside in different functional blocks and different bit slices. The algorithm used to route the over-cell wiring segments for this type of net is described as follows. The algorithm starts by determining two terminals to be connected along an inter-bit net. For the left terminal, the algorithm projects line-probes from all of its legal pins along tracks to the right until an obstruction in each track is encountered. In this situation, an obstruction in a track is a wiring segment of another net. If there is already one or several wires connected to a legal pin from the right, the projecting process is in fact to extend it further to the right. From all of these probed wires, choose the longest one. The same idea is applied to the right terminal to project line-probes to the left, and then find the longest wire as well. The dashed lines in Figure 3.11(a) are the line-

probes generated. If the two longest wires have overlap in the horizontal direction and there exists a channel or channels in the overlapping region, choose a channel if it is the only one, or choose one of them if it has already been formed from the previous routing actions or it is a routine channel. Finally, the longest right-probing and left-probing wires are cut at the left and right edge of the channel, respectively, as shown in Figure 11(b).

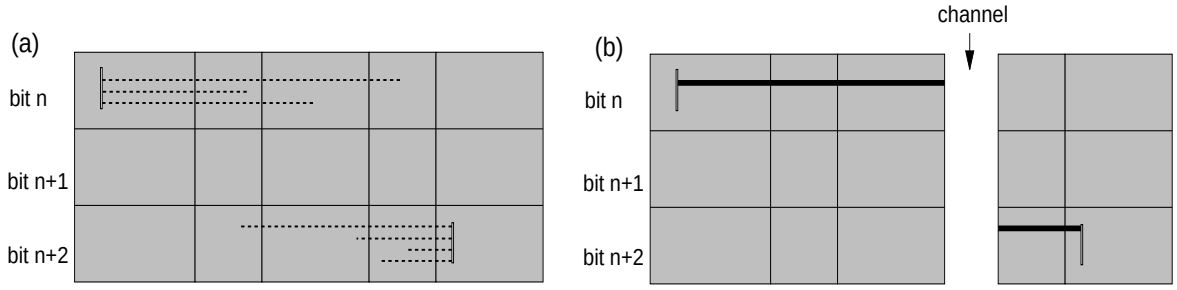


Figure 3.11 *Routing an inter-bit net by projecting line-probes to a channel*

In summary, the over-cell routing algorithm is described as follows:

```
Stretch: For(initial height factor h specified by user;; h++) {
    For(each bit slice, i) {
        For(each track, j) {
            Construct an interval graph G for the net segments on track j;
            Find MIS for G;
            If (bit slice i has been routed)
                break;
        }
        If(bit slice i has not been rourted)
            continue Stretch;
    }
    Route inter-bit nets;
    If (the inter-bit nets routing is successful)
        break;
}
```

3.5.4 Over-cell Routing Algorithm Complexity

In the over-cell routing algorithm described above, most of the time is spent in routing intra-bit

and control nets because generally inter-bit nets only constitute a small fraction of the nets in a datapath. Thus we only analyze the complexity of the over-cell routing in a bit slice. Constructing an interval graph for one track is an $O(n^2)$ operation in the average or worst case; and the complexity of the algorithm used to find the MIS for the interval graph is O(nlogn), where n is the number of pins on the track. Consequently, the overall complexity is $O(n^2)$ for the routing on one track. Assume there are k tracks over a bit slice, hence in the worst case to route an entire bit slice, the complexity is $O(k \cdot n^2)$.

3.5.5 Dogleg Channel Routing Implementation

When the over-cell routing is finished, vertical channels have already been formed with pins on their two edges. The dogleg channel routing algorithm introduced in Section 2.3.1 is employed to complete the connections in these channels. When implementing the algorithm, to check the vertical constraints of a segment that is being placed, every terminal within its range is checked against the pin (if it exists) that is on the other side of the channel. The range defined in the algorithm can be specified by users.

An example result of dogleg channel routing can be found in Figure 3.6(e).

3.6 Features

In this section, we summarize the features of the datapath compiler as follows:

1. It is parameterized in the bus width of a datapath. A set of datapaths with different bus widths is captured as a Java class in the datapath compiler. A datapath with specific bus width can be obtained simply by instantiating an object from the class with parameters specified by users.
2. It is technology portable. Using CLASSIC-SC to generate cells is the main reason that we can achieve this feature in the datapath compiler, because in CLASSIC-SC switching to a new process technology just means writing new technology and physical configuration

files. In the datapath compiler, we have a class that contains a limited number of constants related to process technology. These constants are the necessary information for routing, including the minimum width of a metal 1 layer, the minimum space between two metal 1 layers, the minimum width of a via1 layer, the minimum width of a metal 2 layer, etc.

3. It allows random logic mixed with the bit-sliced structure of datapaths. This feature is accomplished with the aid of relative placement directives in the design capture, and direct support from the routing algorithm. It is important to note that for large cells that are several times higher than standard cells, their heights should be an odd multiple of the heights of standard cells to guarantee the correct abutment of V_{DD} and V_{SS}, whereas this restriction is not imposed by the routing algorithm.
4. It is extensible, especially for the future addition of timing optimization. Much attention has been paid to design the datapath compiler based on the object-oriented approach for future extension. As we will show in the next Chapter, a router implementing a new algorithm might be introduced into the system without affecting other parts of the system. For timing optimization, the system has been designed to provide support for this issue, as discussed earlier in this Chapter. Static timing analysis and timing optimization can be incorporated into the system without affecting the existing design and implementation.

3.7 Summary

In this Chapter, a datapath compiler is introduced in detail in terms of its design capture, cell design, placement, and routing. In addition to the features listed in the last section, the datapath compiler does its cell design using CLASSIC-SC, and features a new type of over-cell routing approach. The flexibility that CLASSIC-SC provides is the source of most ideas, and the object-oriented design makes the implementation process much easier. In the next Chapter, we will present the object-oriented design of the datapath compiler.

CHAPTER 4

The Object-oriented Design of the Datapath Compiler

This chapter describes the objected-oriented design (OOD) of the datapath compiler. We mainly rely on the *Unified Modeling Language (UML)* [Rum99], a visual modeling language primarily used with object-oriented analysis and design methods, to convey the ideas behind our OOD and describe the structure of the software system we developed. For some crucial design decisions, brief discussions will be given. First in Section 4.1, we describe the behavior of the datapath compiler from the computation and workflow perspective using an activity diagram. Then, in Section 4.2, we discuss the object structure of the system using class diagrams, and detail its behavior using collaboration diagrams. Finally, Section 4.3 summarizes this chapter.

To aid the reader in understanding the contents of this chapter, necessary information about *UML* is given in Appendix A of this thesis.

4.1 The Dynamic Behavior of the Datapath Compiler

In UML, there are several modeling constructs, generally called *views*, to represent various aspects of a system: static view, use case view, interaction view, state machine view, activity view, physical view, and model management view. To depict the big picture of how the datapath compiler works, we capture its dynamic behavior using an activity view. The activity diagram is shown in Figure 4.1.

In the diagram, the activities are arranged into vertical zones separated by dashed lines. Each zone represents the responsibilities of a particular role in the system. Note that the role of a user is also

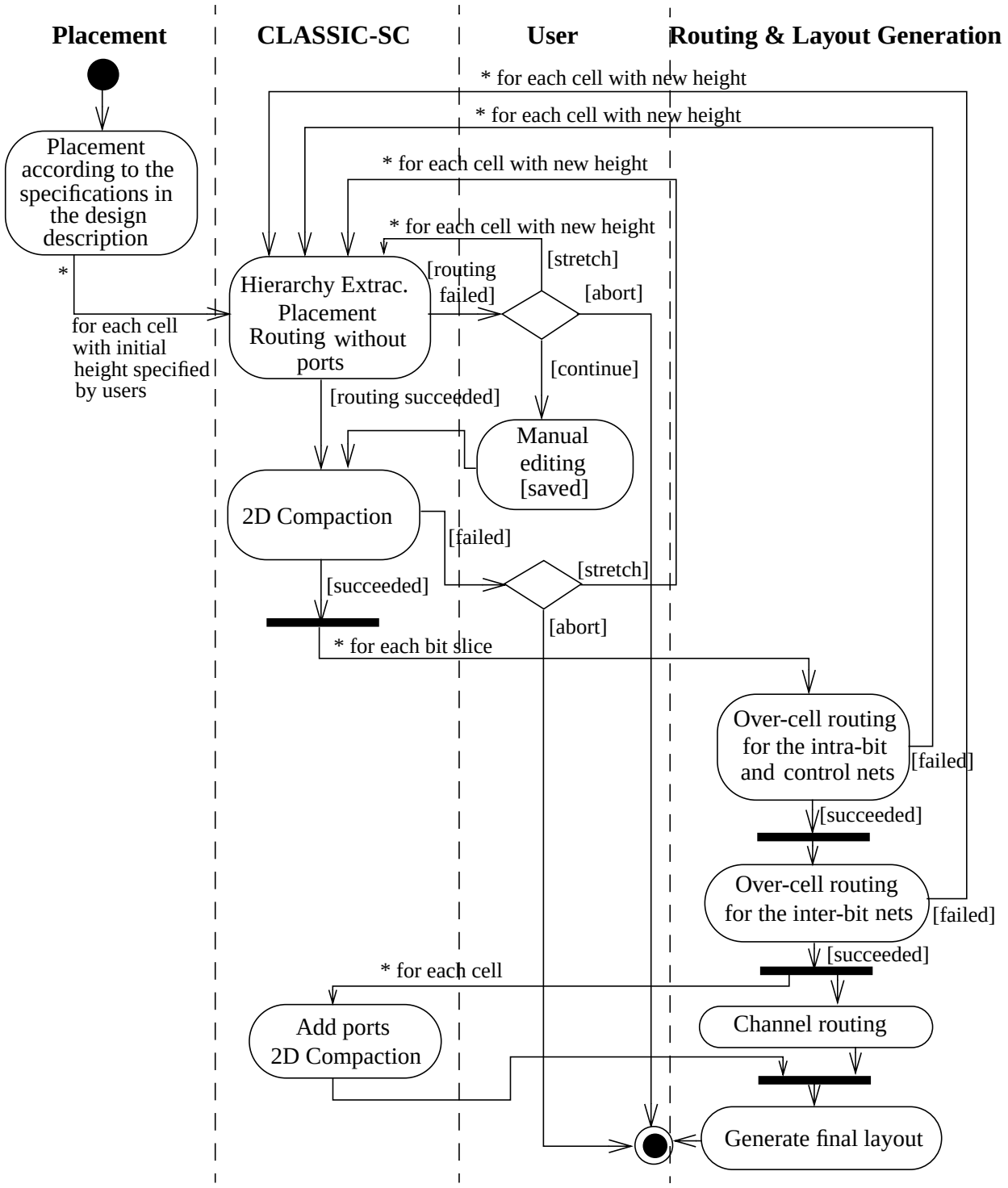


Figure 4.1 *Activity Diagram of the Datapath Compiler*

included in the diagram because, for some cells, CLASSIC-SC may give incomplete-routed results after the routing phase, or infeasible results after the compaction phase, therefore, the user's intervention is needed with the aid of the GUI to deal with this situation so that the whole

system can work properly.

4.2 Object Structure and Object Interactions

In an object-oriented system, the constituent classes and their relationships form the object structure of the system, which is generally described using class diagrams in UML. The classes that constitute the core of the datapath compiler can be grouped into three packages: the classes for cell layout generation, the classes for design capture, and the classes for routing and layout generation. Accordingly, the object structure of the datapath compiler will be introduced in the following three subsections using class diagrams. Although drawing class diagrams is indispensable to describe an object-oriented system, class diagrams are in fact the static view of the system because they do not reflect time-dependent behavior. The behavior of an object-oriented system is implemented through the object interactions by exchanging messages among the objects involved. Therefore, in each subsection, along with the class diagram, we also provide its corresponding collaboration diagram showing the flow of control across the objects which are involved to implement a specific task.

Before proceeding to the subsections, we would like to make it clear that we draw class diagrams from the implementation perspective as we are describing a system that has already been implemented. Generally, there are three perspectives we can use in drawing class diagrams: conceptual, specification, and implementation. Different perspectives affect the way we interpret a class diagram.

4.2.1 Classes for Cell Layout Generation

Classes involved in this task include *SCellPool, CCellPool, Extractor, TsmcClassicSC, GDSIIImporter, TsmcIOGrid*. The class diagram for this part of the system is shown in Figure 4.2. In the diagram, a class is shown as a solid-outline rectangle with three compartments separated by horizontal lines. The top compartment holds the class name; the middle compartment holds a list of attributes; the bottom compartment contains a list of operations. *TsmcClassicSC* is the class

that deals with CLASSIC-SC directly, and for the other parts of the system it plays the role of a cell generation tool. *GDSIIImporter* is used to read in the layouts of the cells, which are exported by CLASSIC-SC in GDSII format. *TsmcIOGrid* provides all information regarding the over-cell tracks. *Extractor* plays the major role in this subsystem. It extracts legal pins for all the ports of a cell after the first compaction. *SCellPool* together with *CCellPool* act as a facade that offers other parts of the system a simple interface to this subsystem, and furthermore, they both serve as a library of the extracted views of legal pins for the cells. As we can see from the cell design procedure introduced in Section 3.3, obtaining a cell's legal pins is a time-consuming process with many interactions between the datapath compiler and CLASSIC-SC. However, for a type of cell with a specific height, their layouts (without ports) coming from CLASSIC-SC are always the same, and so are their legal pins. Therefore, *SCellPool* and *CCellPool* are designed to keep the extracted view of a cell's legal pins every time when a new type of cell is produced. Whenever a request regarding a cell's legal pins comes from the outside, *SCellPool* or *CCellPool* checks with itself first, and if the requested information is found, it sends it out immediately. In this way, the system saves much time because generally a datapath contains a large number of identical cells. *SCellPool* is used for standard cells, while *CCellPool* is for the cells that are designed manually, and whose heights are multiples of the standard cell height. Finally, to make it work the way we plan, *SCellPool* and *CCellPool* are accessed as static members from other classes, which means there is only one instance per class, rather than one in every object created from the class.

Note that in the diagram, *ClassicSC*, *Importer*, and *IOGrid* are interfaces with classes *TsmcClassicSC*, *GDSIIImporter*, and *TsmcIOGrid* as their implemented classes, respectively. An interface is defined as a class with no implementation, and hence, has operation declarations but no method bodies and no fields. Programming to an interface makes possible the flexibility and extensibility we desire, because it gives us a way to establish object connections and message-sends to objects in any class that implements a specific interface, without hardwiring object connections or hardwiring message-sends to a specific class of objects. For example, in our case if CLASSIC-SC can export layouts in another format, a new layout importer could be introduced into the datapath compiler to read in this format without affecting the implemented part of the software as long as the class of the new importer implements the interface *Importer.* As well, a

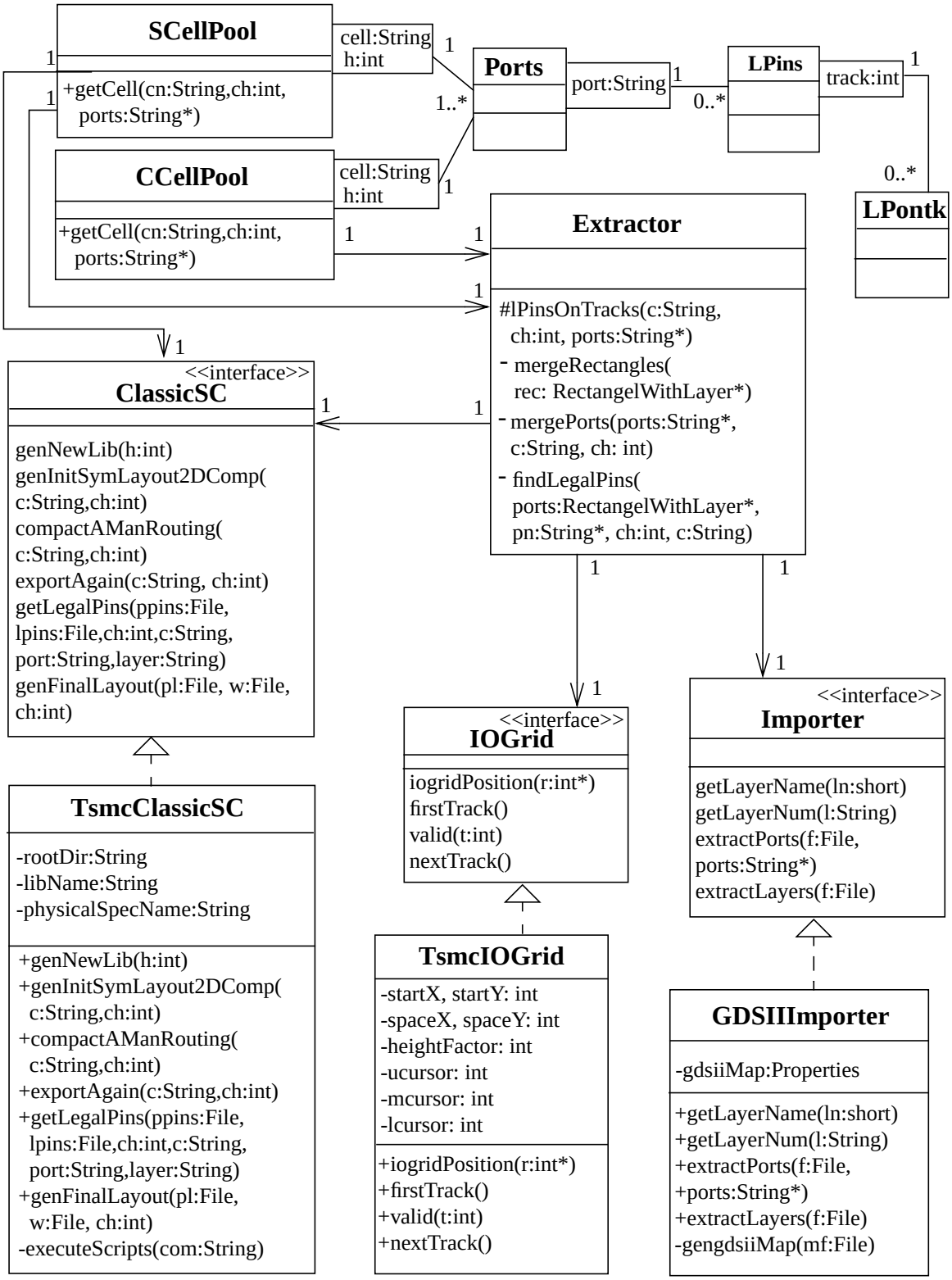


Figure 4.2 *The class diagram of the classes for cell layout generation*

new technology could be integrated into the system as long as the corresponding class implements the interface *ClassicSC*. In the existing system, class *TsmcClassicSC* implements *ClassicSC* to work for the TSMC-0.35 CMOS technology. The same idea is applied to all of the interface-related implementations in our system.

This subsystem is primarily used to implement two tasks: to generate cells (without ports) from netlists and then extract their legal pins, and to produce the final layouts of cells with ports. Their collaboration diagrams are shown in Figure 4.3 and Figure 4.4, respectively. These diagrams show how the objects interact. Note that the first request arrives from the other parts of the system, which is not shown in the diagrams. In Figure 4.4, class *LeafCell*, which is not in this subsystem, is incorporated into the diagram to make the illustration more clear. The *LeafCell* class is described in Section 4.2.2.

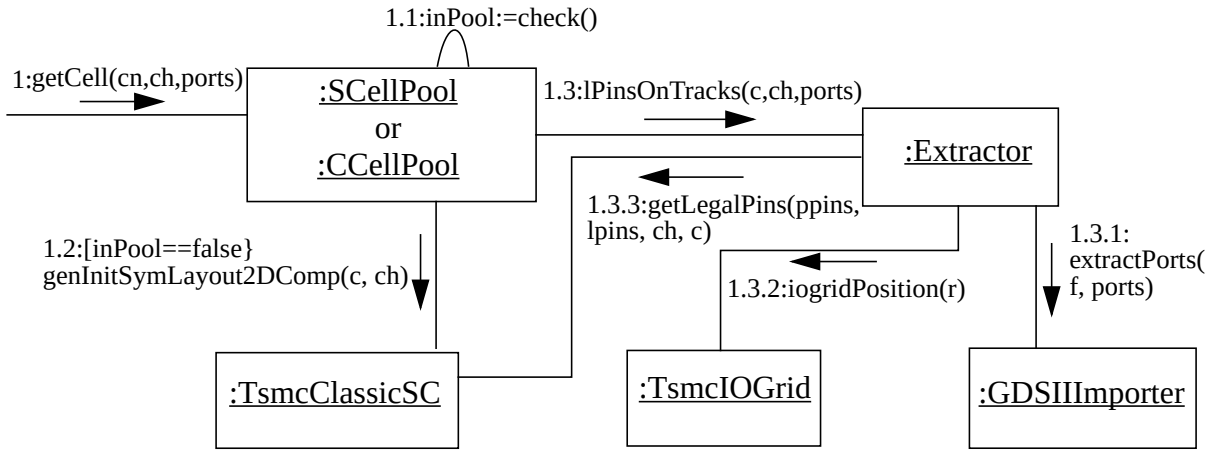


Figure 4.3 *Collaboration diagram for extracting legal pins*

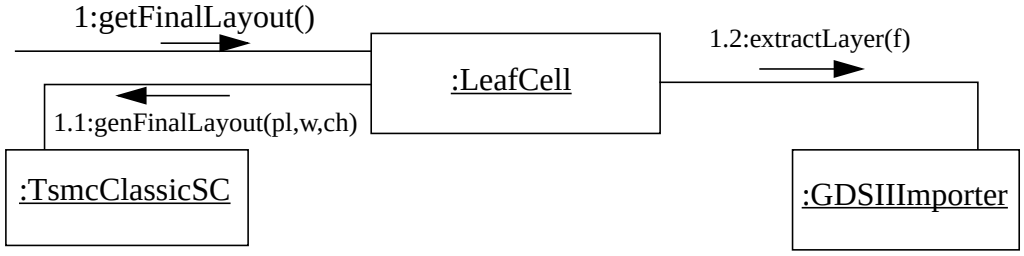


Figure 4.4 *Collaboration diagram for final layout generation*

4.2.2 Classes for Design Capture

As mentioned earlier, classes in this subsystem make up the framework that serves as the foundation for the whole system. The class diagram of this framework is shown in Figure 4.5. Due to lack of space, the query methods used to retrieve the attributes of classes are not shown in the classes' rectangles. *DataPath* is used as the superclass of all the datapath classes created by the user. *FuncBlock* is employed to capture the functional blocks. It is in fact a subclass of *DataPath*, with extra placement information. Like *DataPath, LeafCell* is used as the superclass of all the cell classes written by the user. *CustomColumn* and *RegColumn,* which implement interface *Placement,* are used as relative placement directives. For the usage of the above-mentioned classes, please refer to Section 3.2. *Net* is used to represent interconnections in a datapath. *PortInCell* and *PortInDP* are employed to represent the IO ports of cells in the context of a cell and a datapath, respectively. *PinWithLayer* is the class of physical pins with (x, y) coordinates and layers. Note that in the diagram, to avoid repetition, the classes introduced in the previous diagram are simply drawn as rectangles with their names. However, the symbols of class *Router2Metal* and *LayoutGenerator* are also simplified because they are relatively large classes and will be detailed in the next subsection.

Note that the classes of this subsystem do not participate in any activity directly. Only when used in a real datapath design, will they be involved in a specific activity, such as routing and layout generation. Therefore, there is no collaboration diagram corresponding to the class diagram in Figure 4.5. Class *Datapath*'s role in a collaboration behavior will be described in the next subsection when the classes for routing and layout generation are discussed.

4.2.3 Classes for Routing and Layout Generation

Figure 4.6 shows the class diagram of the routing and layout generation subsystem of the datapath compiler. *Router2Metal* is the main class of this subsystem, which implements the algorithms of over-cell routing and channel routing. Interface *Router* defines a router's operations for future extension, while *Router2Metal* is its existing implementation. *LayoutGenerator* is used to add ports and generate layout for each cell after the routing phase is finished, and then calculate the

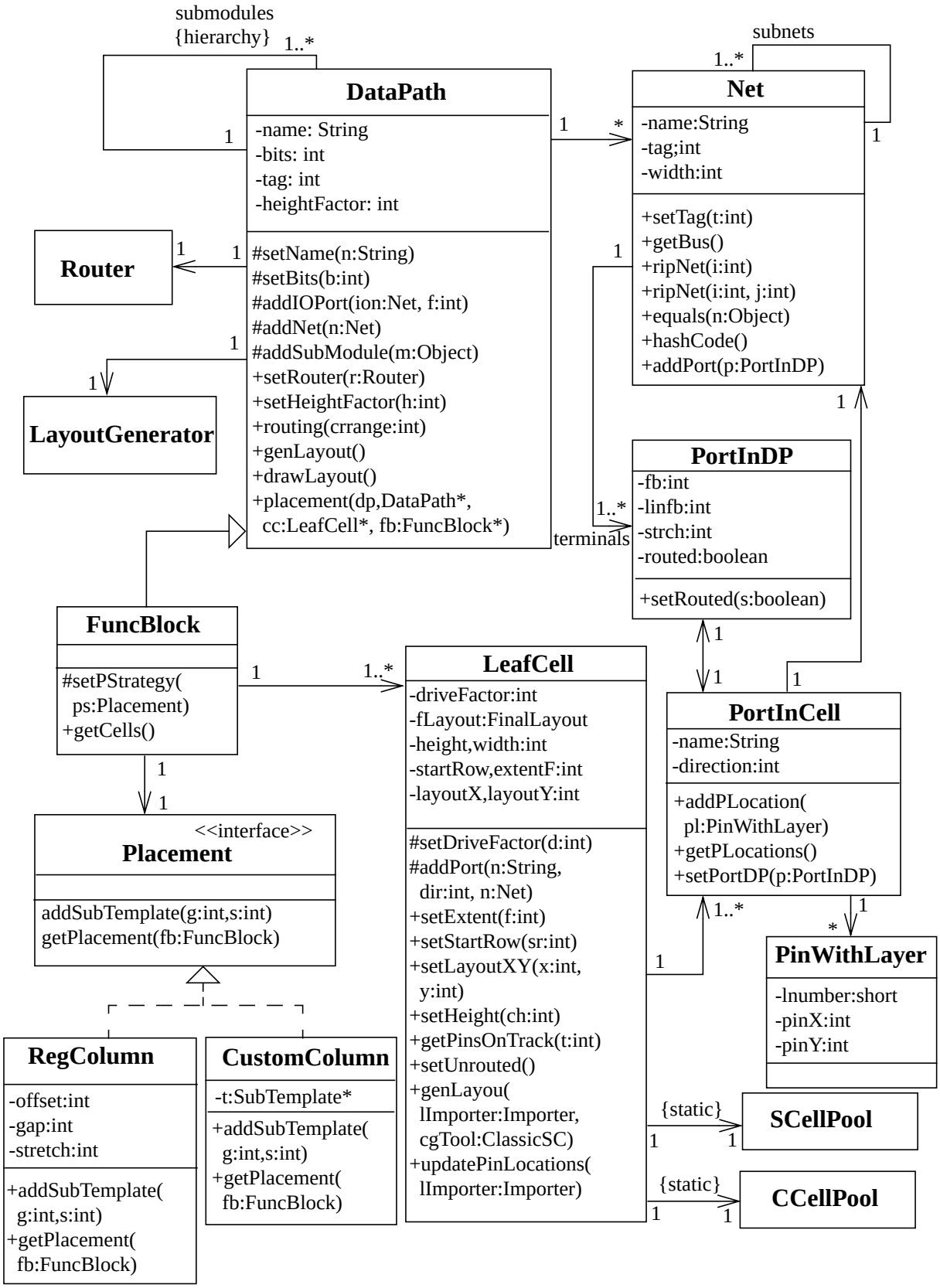


Figure 4.5 *The class diagram of the classes for design capture*

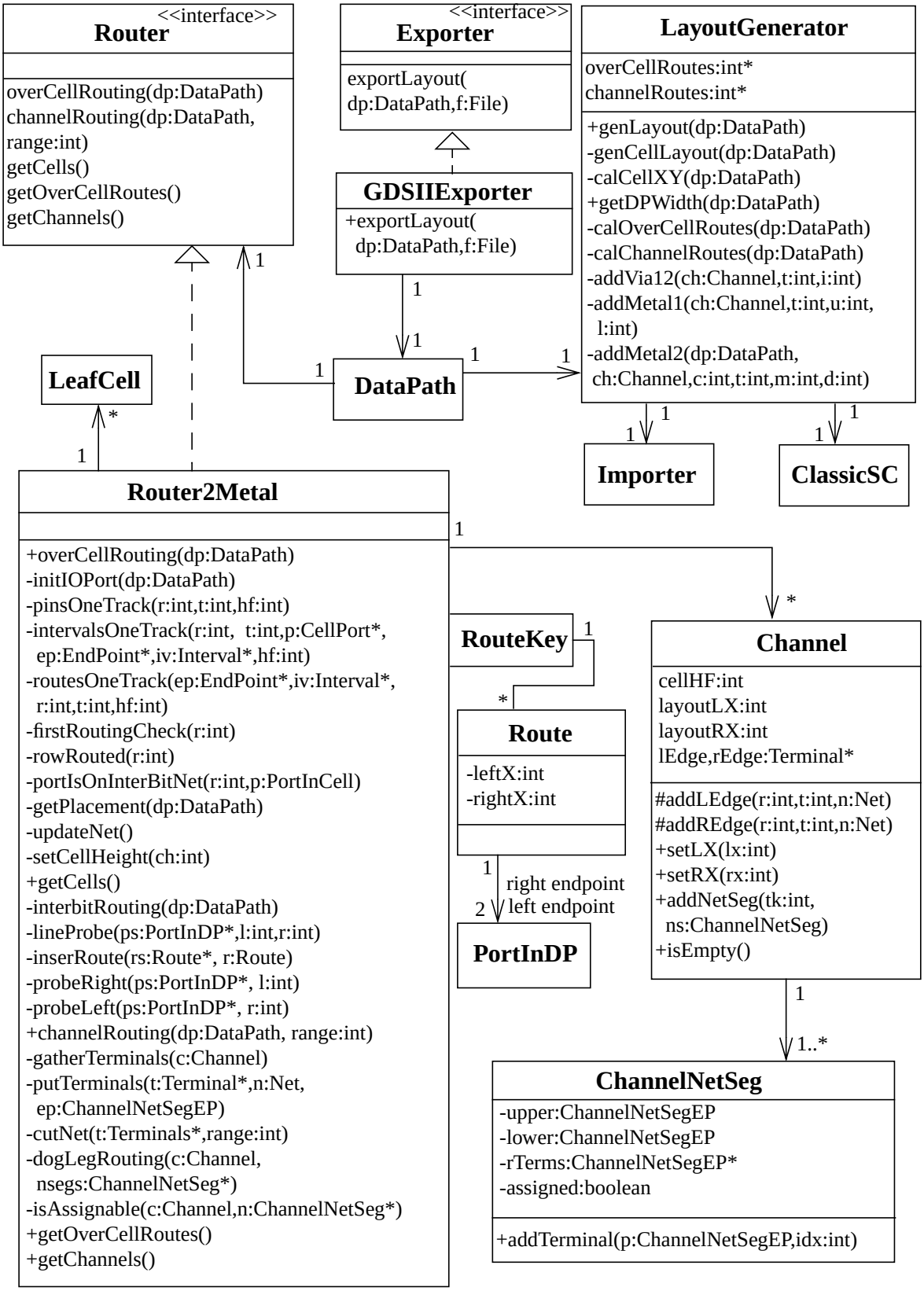


Figure 4.6 *The class diagram of the classes for routing and layout generation*

absolute placement, and finally produce the final layout for a datapath. *GDSIIExporter*, which implements interface *Exporter*, is employed to export layouts in GDSII format. Its private methods are not listed for lack of space. *Route* is the class of all over-cell wiring segments; *ChannelNetSeg* is the class of all the routes in channels. Class *Channel* is used to represent channels in a datapath.

The collaboration diagram depicting the subsystem's behavior in routing and layout generation is shown in Figure 4.7. Note that the first request for layout generation comes from the GUI, which is not shown in the diagram.

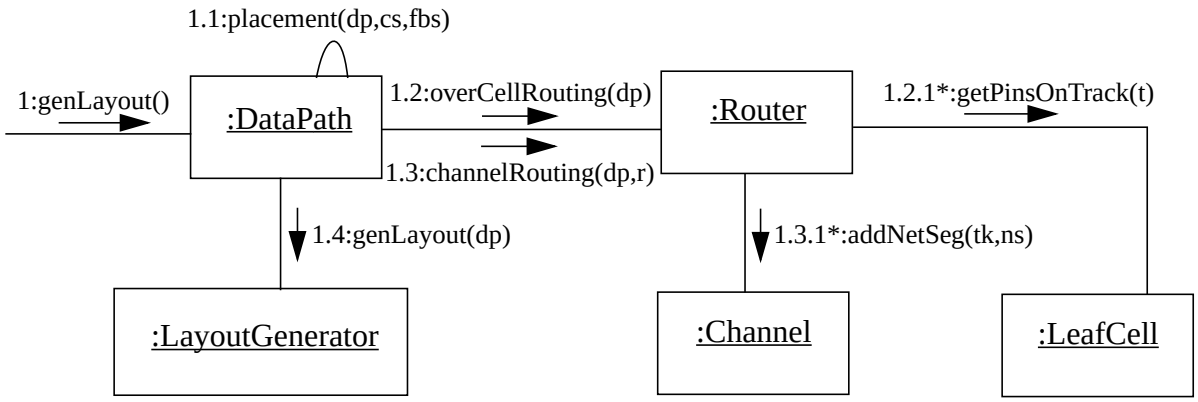


Figure 4.7 *Collaboration diagram for routing and layout generation*

4.3 Summary

In this chapter, the object-oriented design of the datapath compiler is described using UML from the implementation perspective. The dynamic behavior of the datapath compiler is shown in an activity diagram followed by three subsections where the object structure and object interactions of the whole system are introduced in detail by using class diagrams and collaboration diagrams. Note that not all classes but only those significant for representing the object structure of the system are shown in the class diagrams. The technique of programming to interfaces is exploited to design reusable and extensible object-oriented software. The classes for design capture are designed such that all necessary information, especially the netlist of a datapath, can be obtained

for later use by a router.

CHAPTER 5

Implementation and Experimental Results

This chapter introduces the implementation of the datapath compiler, and presents several experimental results produced by the datapath compiler. In Section 5.1, we discuss some implementation issues associated with CLASSIC-SC. We present what has been done and what needs to be done in CLASSIC-SC to aid our datapath compiler in producing high-quality layouts. In Section 5.2, we mainly discuss the GUI-based implementation of the system. In Section 5.3, three typical datapath layouts generated by the datapath compiler are given. To evaluate the area and timing of the layouts, the results are compared with those produced from either a standard-cell based synthesis design flow or manual layout. Finally, Section 5.4 summarizes this chapter.

5.1 Implementation Issues with CLASSIC-SC

As introduced in Section 3.3, CLASSIC-SC has been used in our system to generate cell layouts on-demand. The flexibility that CLASSIC-SC provides in technology portability and cell architecture specification is the main reason that we can use it for our application. By writing corresponding technology and physical specification files, we have successfully incorporated TSMC 0.35-micron CMOS technology into CLASSIC-SC and made it work for our experimental datapaths. However, CLASSIC-SC is primarily designed to create standard cell libraries used for traditional ASIC system design. Making it work in our datapath compiler requires some special considerations. The implementation issues related to CLASS-SC that we have encountered and resolved, and the issues we hope to be solved more effectively in the future are discussed below.

CLASSIC-SC is designed as an interactive tool that puts the cell design process in control of the users. It is not uncommon that during the cell generation process CLASSIC-SC may give incompletely-routed results after the routing stage, or infeasible results after the compaction phase. In a stand-alone CLASSIC-SC session, the user may be required to complete the routing manually when an incompletely-routed result is obtained, or redo the design from any stage appropriate when an infeasible result occurs during compaction. However, in our application we require that CLASSIC-SC provide fully-routed feasible cell layouts without any interruption. To deal with these situations, the GUI (which will be introduced in the next section) of our datapath compiler is designed to facilitate the user's intervention, as described below.

Whenever an incompletely-routed cell layout is obtained, the thread running the layout generation process at the background is suspended, and then the user is informed about this situation through the GUI. The user has the following three choices at this point:

1. Go to CLASSIC-SC to complete the routing manually, save the result and notify the suspended thread to continue.
2. If the manual routing fails, or without any attempt to complete the routing manually, inform the system to continue by stretching cells, or
3. Abort by stopping the thread immediately.

The same scheme is applied when an infeasible cell layout is obtained after the compaction, except that there is no way to salvage an infeasible layout in CLASSIC-SC, and therefore, the user has only two choices: to let the system continue by stretching cells or abort.

To date, GDSII Stream data format, which is most commonly used for electron beam lithography and photomask production, is the only format in which CLASSIC-SC exports cell layouts. A GDSII stream file generally contains geometrical objects such as polygons and paths of various layers. Hence the names of nets that are associated with I/O ports generally cannot be found in the GDSII files. However, for routing purposes we need this information in the datapath compiler. To solve this problem, we have modified the behavior of the GDSII exporter of CLASSIC-SC to export the net name along with every polygon of the net, and accordingly the GDSII importer of

our datapath compiler is designed to adapt to this modification.

As mentioned in Section 3.5.3 when we discussed the over-cell routing strategy, the center region of a cell between the p- and n-transistors has the most choices for IO ports. Therefore, to get more routing resources in terms of more legal pins, it is desirable to make the center region as large as possible. However, the compactor in CLASSIC-SC tends to move p- and n-transistors to the center, we think, due to the fact that CLASSIC-SC is designed to generate cells for standard cell libraries used in the traditional standard-cell based synthesis design methodology. By observing the behavior of CLASSIC-SC's compactor, currently we can only keep the center region open by defining a non-uniform routing grid near the VDD power rail in the physical specification file. This keeps the p-transistors at the top of a cell and n-transistors at the bottom. We only regard this approach as a partial solution because for some cells the compactor still does not work the way we hope. We would expect to get a full control over this situation if CLASSIC-SC provided direct support for this kind of use in its compaction strategy.

The cell design using CLASSIC-SC in our approach is composed of two compactions, as described in Section 3.3. It is important that in the second compaction the via1 holes (holes between M2 and M1) can only be moved in the horizontal direction. This is mandatory because the track that a via1 should stay on is imposed by the over-cell router prior to the second compaction. We have achieved this goal by modifying a compaction constraint callback function in a shape component file associated with the I/O ports.

One issue that is important but we still cannot control in our datapath compiler is that after the first compaction, it is not guaranteed that we can get at least one legal pin for an I/O port, mainly because the compactor in CLASSIC-SC is supposed to work on layouts with placed I/O ports. For a complex datapath consisting of many different types of cells, which furthermore could also be very complex with many transistors and I/O ports, this deficiency could manifest itself as a fatal flaw. In fact, we did suffer from it when we worked on the experimental datapaths, which will be discussed in Section 5.3. We hope that in the future we can get direct support from CLASSIC-SC so that both its router and compactor are aware of this kind of use and leave enough space at the two sides along the paths of nets associated with I/O ports.

5.2 GUI-based Implementation

As mentioned earlier, the object-oriented design of the system makes the implementation process much easier. The main part of the datapath compiler is implemented with the *Java 2 Software Development Kit (SDK)*, with a few cell generation scripts written in the *AL* embedded language of CLASSIC-SC. In this section, we briefly introduce the implementation of the GUI of the system.

The motivation for us to develop a GUI for the system is to show the result — layouts generated by the datapath compiler. A snapshot of the GUI is shown in Figure 5.1. The GUI is developed based on the *Java Foundation classes (JFC)*, from which *JFC/Swing* is used as the user interface toolkit to develop the GUI, and *Java 2D API* is used as the drawing toolkit to draw the layouts of datapaths.

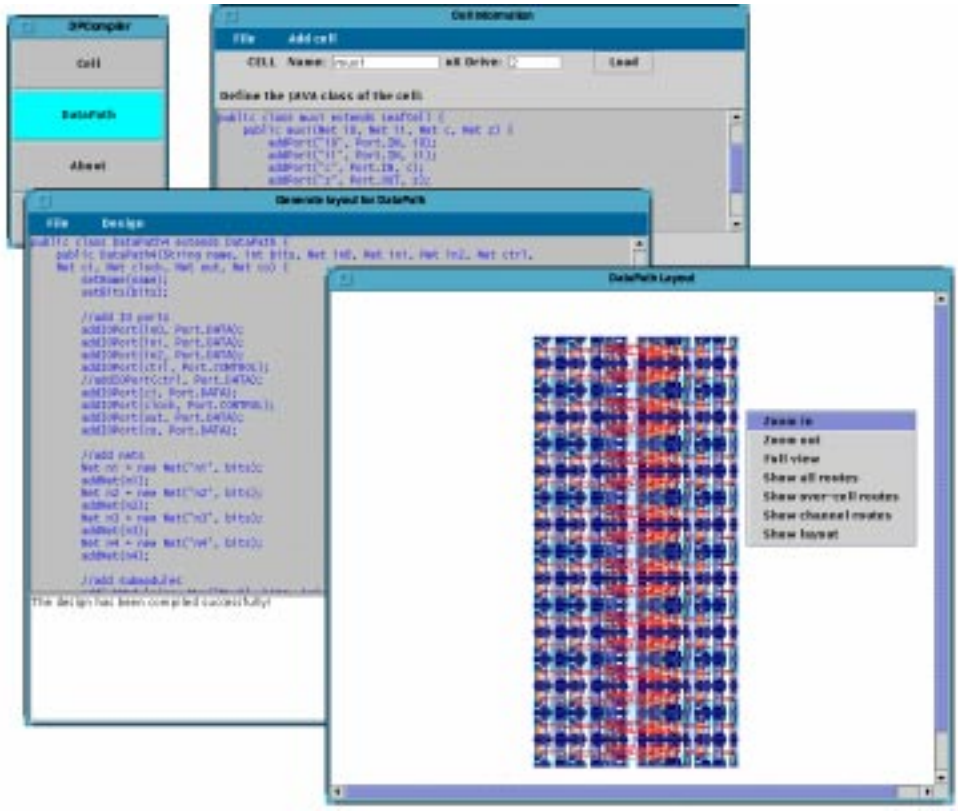


Figure 5.1 *A snapshot of the GUI*

In addition to the function of displaying layouts, the GUI also provides an interface for capturing

designs and exporting layouts. To import a new type of cell into the system, the user is requested to type in the necessary information of the cell, including its Java class entity and transistor-level netlist. For a cell with a different drive strength, if its Java class is already in the system, only its netlist is required. Similarly, a datapath can be captured and compiled in the GUI. For various datapath designs, the GUI is able to check its parameters, and then prompt a dialog window accordingly to let the user specify its data bit width and the bus width for each IO signal. For the detailed information on how to use our datapath compiler, please refer to Appendix B.

The initial version of the GUI suffered a speed problem when displaying and manipulating (zoom in, or zoom out) the layout. We found that this problem occurred mainly because we were using the *reference implementation* of Java 2 SDK. By using the *production release* of Java 2 SDK, which is bundled with a Just-In-Time (JIT) compiler, the performance improved significantly. The speedup is about 40% to 60%. The performance problem is further alleviated by eliminating the alphacomposite, which provides transparency composition in rendering operations in Java 2D API, when drawing the layout. However, the drawing speed is still not satisfactory compared to those developed based on the X-window development kits. It is known that Java applications are still a bit slow even though the performance in Java applets is no longer an issue.

5.3 Experimental Results

To test the performance of our datapath compiler, we used three datapath examples. In one case, we had a full custom layout available; in the other two cases, we used a standard-cell based synthesis flow to generate layouts for comparison to the results produced by our datapath compiler. Our focus for this work was on the resulting area, but we did a simple experiment to see how the timing would compare.

5.3.1 Experimental Datapaths

Our datapath compiler has been used to generate layouts for three typical datapath designs: DP1, Drcp, and Mult4×4.

DP1 is a typical datapath whose schematic is shown in Figure 5.2. It consists of three n-bit registers, three multiplexers, an n-bit ALU, and two n-bit tri-state buffers. The layout of its 16-bit implementation is shown in Figure 5.3.

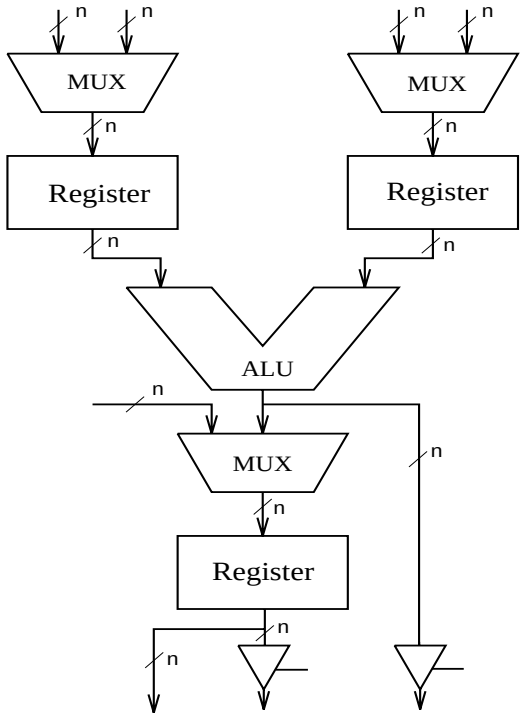


Figure 5.2 *Schematic of DP1*

Drcp is part of a datapath design provided courtesy of James Goodman. It comes from an energy-efficient, scalable encryption processor project at MIT [Goo99]. The datapath is the main part of an encryption engine that uses an algorithm known as the Quadratic Residue Generator to generate a cryptographically-secure pseudorandom keystream sequence that is then XORed with a serial data stream to form the encrypted data stream. One bit slice of Drcp consists of eighteen 2-to-1 multiplexers, five D flip-flops, two AND gates, and two 1-bit Full Adders. The simplified schematic of Drcp is shown in Figure 5.4. We received transistor-level schematics of one bit slice from James, from which the layout of its 8-bit implementation is generated by the datapath compiler, as shown in Figure 5.5.

Mult4×4 is a 4×4 carry-save multiplier whose block-level schematic is shown in Figure 5.6. In fact, this type of array multiplier is not the kind of datapath our system favors because almost half

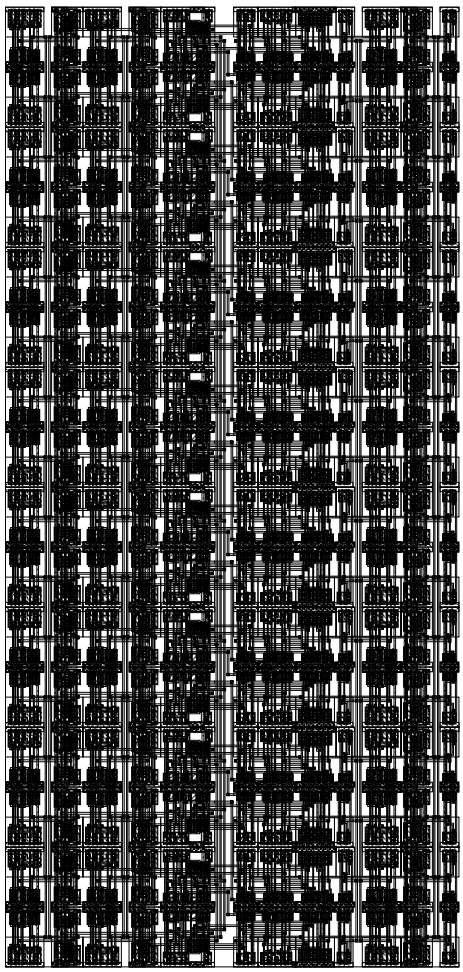


Figure 5.3 *Layout of 16-bit DP1*

of its interconnections are inter-bit nets. However, we use it as an example mainly because multipliers are important components in datapaths, and the critical timing path in this type of multiplier is easily identified so that we can use it as an example circuit to evaluate the timing of the layouts generated by the datapath compiler. The layout of Mult4×4 is shown in Figure 5.7.

The characteristics of the three datapaths as produced by our compiler are summarized in Table 5.1. In the table, the number of transistors and the number of bits in each datapath are shown in *Transistor counts* and *Bit width*, respectively. The numbers of tracks over each bit slice of datapaths are listed in *Tracks per row.* The resulting area for each datapath produced by the datapath compiler is listed in *Area*.

TABLE 5.1 *Characteristics of experimental datapaths*

Datapath	Constituent components	Transistor counts	Bit width	Tracks per row	Area ($\times 10^4$ square microns)
DP1	ALU, Registers, Multiplexers, Tri-state buffers	2208	16	15	5.64
Drcp	Full adders, Multiplexers, Registers	2584	8	13	5.27
Mult4x4	Full adders, Half adders	488	4	14	1.12

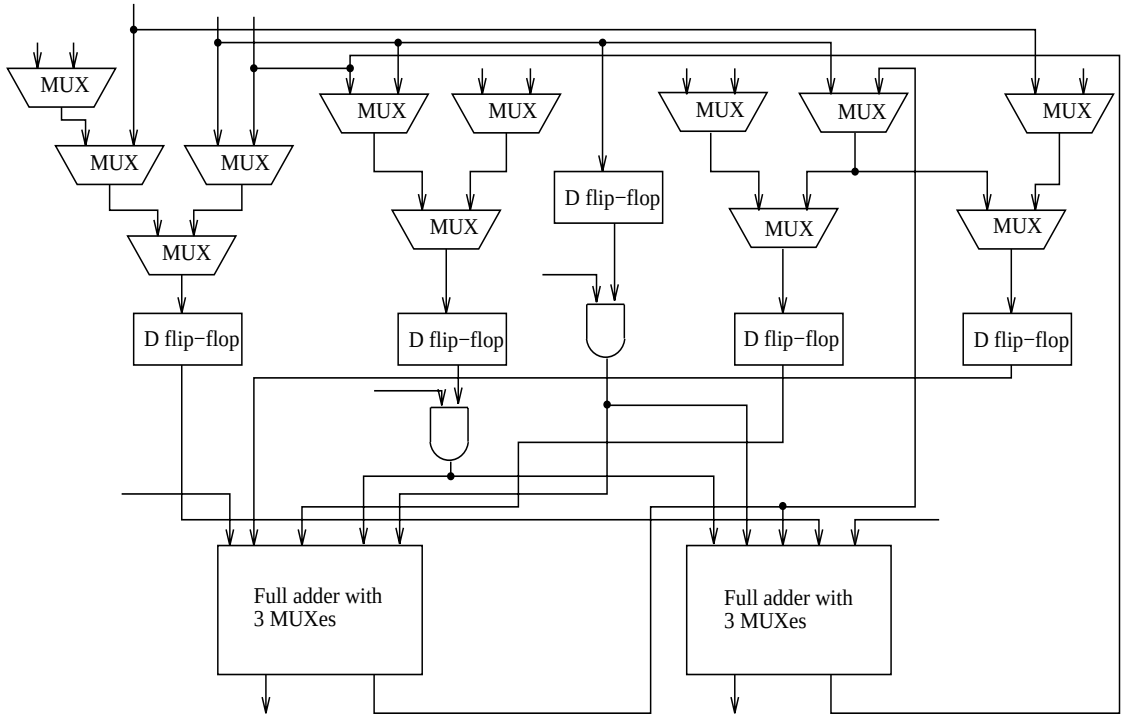


Figure 5.4 *Schematic of Drcp (One bit slice)*

5.3.2 Comparison to other layout methods

To evaluate the results produced by the datapath complier, DP1 and Mult4×4 were also implemented using the standard-cell based synthesis design methodology from VHDL to automatic Placement&Routing. To make the results comparable, the two designs are captured at the structural level of VHDL by instantiating the various components according to the block-level

netlists shown in Figure 5.2 and Figure 5.6, and the sizes of the transistors in the transistor-level netlists of the cells that are imported into our datapath compiler are the same as those of the standard cells used for the synthesis design flow.

In the automatic placement and routing phases, we used *Silicon Ensemble (SE)*, which is an automatic Placement&Routing tool from *CADENCE* that uses area routing, to complete the routing between standard cells. The layouts of DP1 and Mult4×4 are obtained by taking advantage of SE's ability to route without channels. The results show that even though our datapath compiler uses channels, we can still get better areas in our approach.

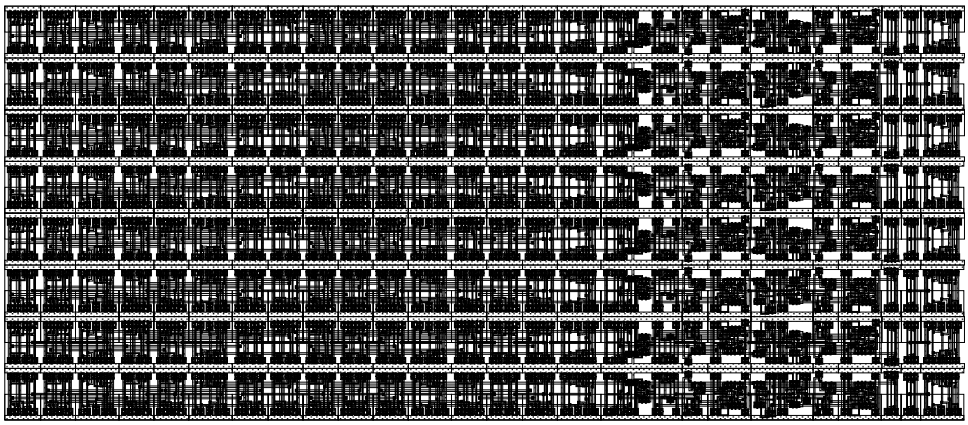


Figure 5.5 *Layout of 8-bit Drcp*

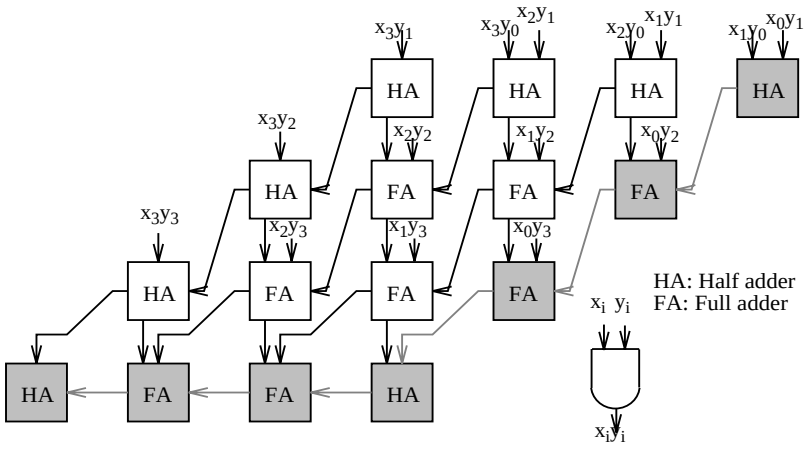


Figure 5.6 *A 4×4 carry-save multiplier* (The critical path is shaded)

For Drcp, the original design was implemented in a full-custom layout using 0.25-micron CMOS

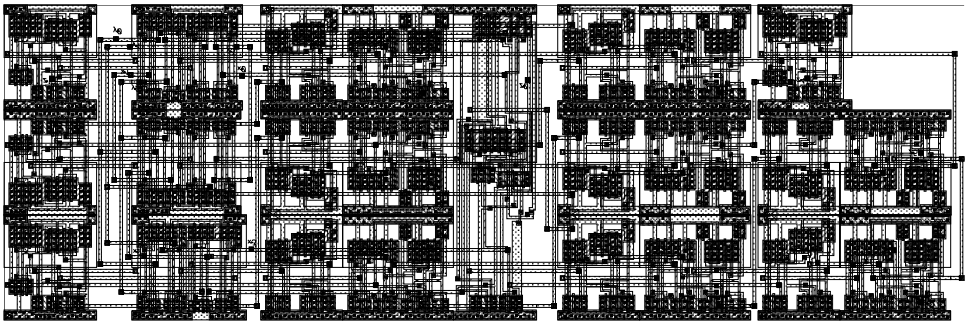


Figure 5.7 *Layout of Mult4×4*

technology. The area of one bit slice of this hand layout is *4500* square microns. The area of our implementation in 0.35-micron CMOS technology is *6590* square microns. We have scaled the transistors in the original design up to the 0.35 feature size by keeping the W/L unchanged for each transistor. We scaled our resulting layout down accordingly to get an approximation of the area in a 0.25-micron technology. The down-scaled area is *4707* square microns, which is only slightly worse than the hand layout. The data listed in Table 5.2 is for an 8-bit Drcp.

Table 5.2 summarizes the results and compares our datapath compiler implementations with the other implementations. For Drcp, the negative number in *Area reduction* shows that our implementation is only 4.72 percent worse than the full-custom layout. For Mult4×4, as mentioned in Section 5.3.1, it is not the type of datapath that our compiler favors because almost half of its interconnects are inter-bit nets, which inevitably results in channels. Consequently, as we can see, we did not get much area reduction in Mult4×4 compared to the layout from SE, which completes routing without channels.

As mentioned at the beginning of this thesis, we did not address the timing aspects of the optimization directly in our datapath compiler. However, in the experiment, we still tried to investigate how the resulting layout performs in timing. Mult4×4 is used as the test example. We use *Hspice* simulation to verify the function of the layout, as well as to measure the propagation delay of its critical path. The propagation delay with a 0.05pF load at outputs is listed in Table 5.2. It shows our implementation is faster than the layout from SE, where we believe the primary benefits are from the regular arrangement in the placement of the datapath compiler.

In fact, using static timing analysis rather than Hspice simulation is the best way to obtain the propagation delay of the critical path of a circuit, mainly because in simulation obtaining the delay information on the critical path is input pattern dependent. However, since the layouts we can get from the datapath compiler are in GDSII format with geometrical polygons of various layers, they do not contain information related to the constituent cells whose timing information is imperative for static timing analysis. Therefore, Hspice simulation is the only way for us to get the timing of resulting layouts. For the above reason, we did not get the corresponding timing information for DP1 and Drcp because long Hspice simulation runs would be required.

TABLE 5.2 *Comparison in area and propagation delay*

Datapath	**Area ($\times 10^4$square microns)**		**Area reduction (%)**	**Propagation delay of critical path (ns)**	
	Datapath compiler	Standard-cell based synthesis, P&R[#] or Manual layout[*]		Datapath compiler	Standard-cell based synthesis, P&R
DP1	5.64	7.58[#]	25.60	—	—
Drcp (8 bits)	3.77 (down-scaled)	3.60[*]	- 4.72	—	—
Mult4×4	1.12	1.14[#]	1.75	2.1	2.4

We do not regard the timing result we get from Mult4×4 as conclusive proof, and expect that the timing aspect can be guaranteed in the future when the static timing analysis and optimization function is added into the system.

5.4 Summary

In this chapter, the experimental results of three typical datapaths have been presented. We show that our datapath compiler can produce better results for datapaths than the traditional standard-cell based synthesis design flow, especially in area. Compared with a hand layout, the result is only a bit worse. We believe these results can be achieved mainly because datapaths are treated specifically by taking advantage of their regular bit-sliced structure. We also attribute this result to

CLASSIC-SC, which we use to generate cell layouts on the fly, and the over-cell routing algorithm we developed for datapaths.

We recognize that timing considerations must still be incorporated into the system before it can be truly viable. However, the ability to generate cells on the fly means that it should be possible to generate the necessary cells needed to meet timing constraints once some timing analysis is incorporated.

CHAPTER 6

Conclusions and Future Work

6.1 Thesis Conclusions

This thesis has presented the datapath compiler we developed to generate layouts for datapaths. A new type of over-cell routing algorithm designed specifically for datapaths has been proposed and implemented. This is the first time that a cell layout generation tool has been used to generate cells on the fly in a datapath compiler. All previously known systems have used pre-defined libraries. Using CLASSIC-SC for cell generation is also the main reason that we can achieve technology portability in our datapath compiler. Other features of the datapath compiler include parameterization in bit width, the ability to deal with the irregularity that exists in real-world datapaths, implementation using an object-oriented design for future reuse and extension, and a GUI for capturing designs, displaying and exporting layouts.

Experimental results show that the datapath compiler can generate high-quality layouts with smaller areas than standard-cell based synthesis and automatic Placement&Routing tools. We believe, by exploiting the regularity in datapaths, the datapath compiler should be able to produce better results in timing as well. Based on the module generation techniques, the datapath compiler has been designed for future enhancement by integrating the static timing analysis and optimization into the system so that better results in both area and timing can be achieved.

6.2 Future Work

As discussed in Section 5.1, some implementation issues associated with CLASSIC-SC suggest

that CLASSIC-SC be capable of use in a datapath compiler and provide direct support at a lower level. The efficiency of our datapath compiler in terms of the CPU time it costs to generate layouts for datapaths could be improved if CLASSIC-SC can be coupled into the system more closely. For example, if we can get cell layouts directly from the inner representation in CLASSIC-SC instead of the GDSII files exported, much flexibility can be achieved and much time can be saved in executing a datapath design.

To make the datapath compiler work as we expected in terms of timing, the next step would be to incorporate static timing analysis and optimization into the system. This may require that CLASSIC-SC be able to provide timing properties for each cell it generates, and require the datapath compiler itself to model the delay on interconnects. The static timing analysis and optimization may be incorporated into the whole process of layout generation in the datapath compiler, including the optimization in the pre-layout stage and to get necessary information for timing-driven routing, and in the post-layout stage to perform the optimization using the techniques such as cell sizing and buffer insertion.

For technologies with more routing layers, future work also includes developing new routing algorithms to get denser layouts.

APPENDIX A

Unified Modeling Language

In this appendix, we will give a brief introduction to the Unified Modeling Language (UML) [Fow97]. The purpose of this appendix is to aid the reader in understanding the contents of Chapter 4, where the object-oriented design of the datapath compiler is described using UML.

UML is a standard object modeling language. It is used to specify, visualize, construct, and document the artifacts of a software system, especially the software developed using object-oriented development methods.

Generally, UML modeling constructs are divided into several *views* to capture information about the static structure and dynamic behavior of a system. These *views* are static view, use case view, interaction view, state machine view, activity view, physical view, and model management view. A system is modeled as a collection of objects that interact with each other to implement a specific task that ultimately benefits an outside user. The static structure defines classes that are important to a system and their relationships including association, generalization, and various kinds of dependency. The dynamic behavior defines the behavior of a system over time and the communications among objects to accomplish goals.

In Chapter 4, the object-oriented design of the datapath compiler is described using the activity view, static view, and interaction view of UML. These views will be introduced in the following subsections.

A.1 Activity View

The activity view of a system is normally shown in activity diagrams. An activity diagram is a special form of state machine intended to model computations and workflows. It contains activity states, each of which represents the performance of an activity in a workflow. Instead of waiting for an event, an activity state waits for the completion of its computation. When the activity completes, execution proceeds to the next activity state within the diagram. A completion transition fires when the preceding activity is complete. An activity diagram may contain branches, as well as forking of control into concurrent threads. Concurrent threads represent activities that can be preformed concurrently by different objects.

In activity diagrams, an activity state is shown as a box with rounded ends containing a description of the activity. Two special states, initial state and final state, are displayed as a small filled black circle and a small filled black disk surrounded by a small circle, respectively. Simple completion transitions are shown as arrows. Branches are shown as guard conditions on transitions or as diamonds with multiple labeled exit arrows. A fork or join of control is shown as multiple arrows entering or leaving a heavy synchronization bar. Figure A.1 shows a typical activity diagram containing all kinds of symbols.

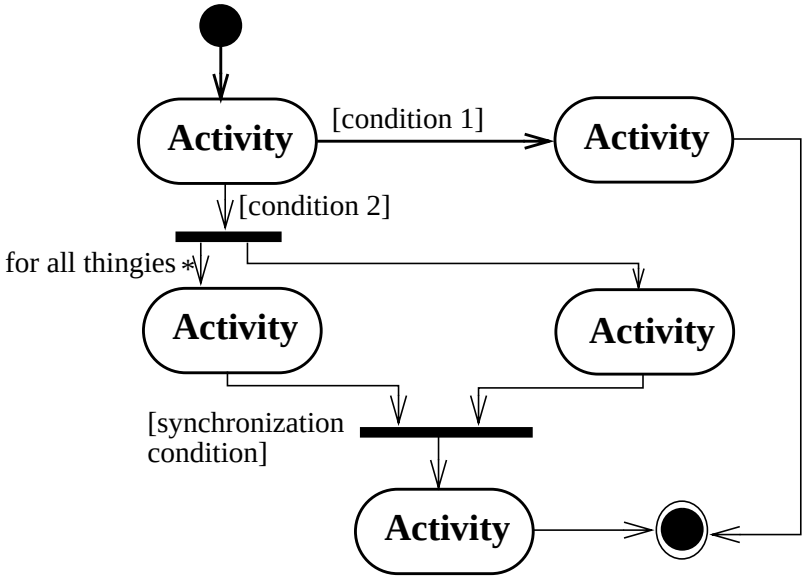


Figure A.1 *An example activity diagram*

It is often useful to arrange an activity diagram into vertical zones separated by dashed lines. Each zone represents the responsibilities of a particular role in a system. Each vertical zone is generally called a swimlane.

A.2 Static View

A static view is a view of the overall model that characterizes the elements in a system and their static relationships to each other. The main constituents of the static view are classes and their relationships: association, generalization, and realization. The static view is displayed in class diagrams.

In class diagrams, classes are drawn as rectangles. Lists of attributes and operations are shown in separate compartments, as shown in Figure A.2. Relationships among classes are drawn as paths connecting class rectangles. The different kinds of relationships are distinguished by line texture and by adornments on the paths or their ends, as detailed in the following paragraphs.

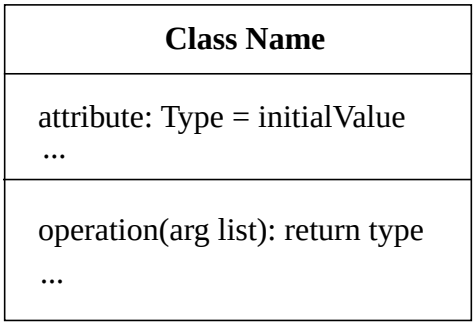
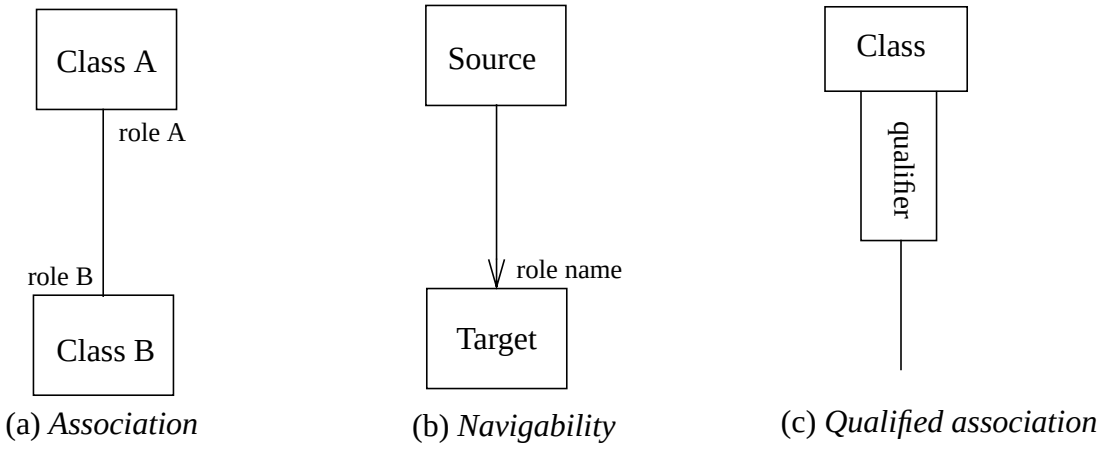


Figure A.2 *Class notation*

The association relationship describes semantic connections among individual objects of given classes in a system. Associations are the “glue” that ties a system together. Without associations, there are nothing but isolated classes that do not work together. The notation of a binary association is a solid line or path connecting the participating classes. The association name is placed along the line with the rolename and multiplicity at each end, as shown in Figure A.3(a). If there is only one association between a pair of classes, the association name and role names may not be necessary because the class names are sufficient to identify the association. During the

design and implementation stages of an object-oriented system, associations capture design decisions about data structures, as well as separation of responsibilities among classes. In this case, directions are always imposed on associations to represent state information available to a class, as shown in Figure A.3(b). An association can also have attributes of its own. If an association attribute is unique within a set of related objects, then it is a qualifier. A qualifier is a value that selects a unique object from the set of related objects across an association. The notation of a qualified association is shown in Figure A.3(c). The multiplicities at the association ends specify the possible number of objects that may be related to an object, and their notations are shown in Figure A.4.



(a) *Association* (b) *Navigability* (c) *Qualified association*

Figure A.3 *Association notation*

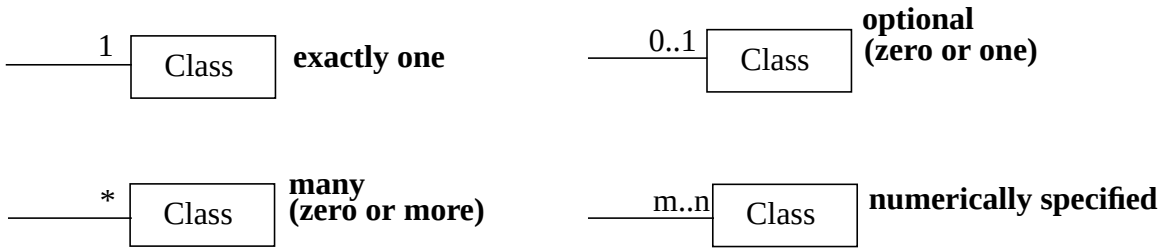


Figure A.4 *Multiplicity notation*

The generalization relationship relates general descriptions of superclasses to more specialized subclasses. Generalization facilitates the incremental description of an element by sharing the descriptions of its ancestors, which is generally called *inheritance*. In UML, a generalization is drawn as an arrow from the subclass to the superclass, with a large hollow triangle on the end

connected to the superclass, as shown in Figure A.5.

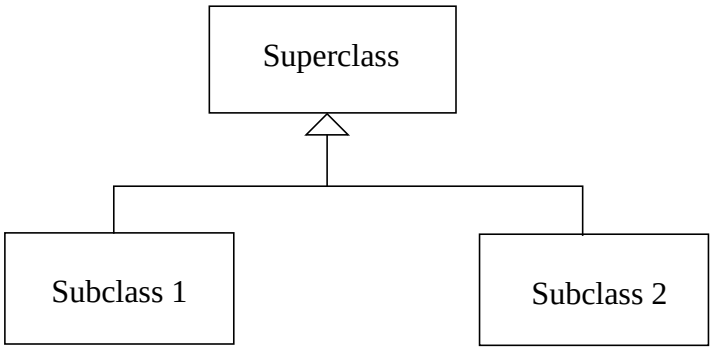


Figure A.5 *Generalization notation*

The realization relationship relates a specification to an implementation. An interface is a specification of behavior without implementation; a class includes implementation structure. One or more classes may realize an interface, and each class implements the operations found in the interface. In UML, a realization is displayed as a dashed arrow with a closed hollow arrowhead connected to the interface, as shown in Figure A.6.

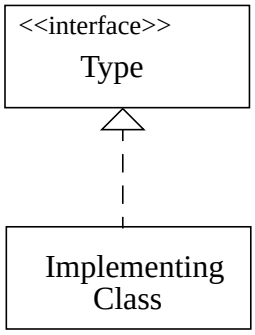


Figure A.6 *Realization notation*

A.3 Interaction View

The interaction view describes sequences of message exchanges among roles that implement behavior of a system. In UML, there are two kinds of interaction diagrams that can be used to display the interaction view: sequence diagrams and collaboration diagrams. In this appendix, we only introduce collaboration diagrams because we used them to describe the behavior of our

datapath compiler in Chapter 4.

A collaboration is the description of a collection of objects that interact to implement some behavior within a context. It describes a society of cooperating objects assembled to carry out some purpose. In UML, collaboration diagrams are used to display the objects and links involved in a collaboration at run time. In collaboration diagrams, an object is shown as a rectangle with the object name in it. The object naming scheme takes the form *objectName:Classname*, where either the object name or the class name may be omitted. A message is shown as an arrow attached to a link line connecting the participating objects. The sequence of messages is indicated by sequence numbers prepended to message descriptions. A typical collaboration diagram is shown is Figure A.7.

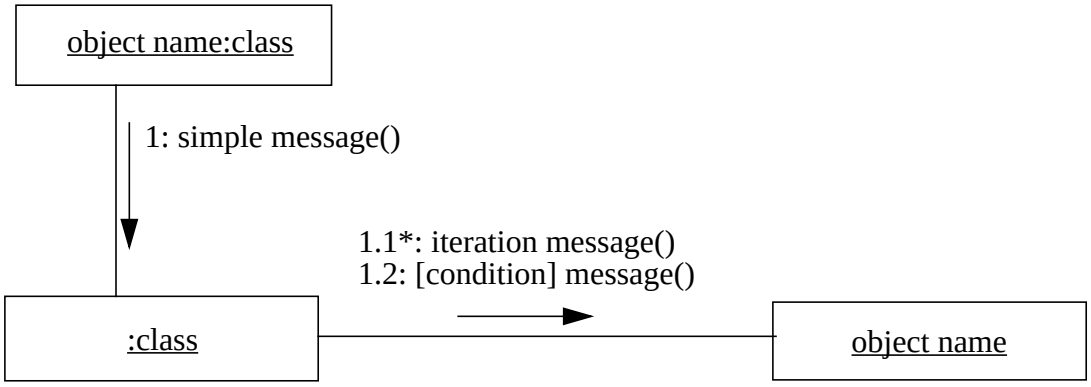


Figure A.7 *An example collaboration diagram*

APPENDIX B

A User Manual for the Datapath Compiler

In this appendix, we provide a short user manual for using our datapath compiler with the GUI. All of the Java program files of the datapath compiler are in directory */pc/e/4/encoder/jiangh_test/java/java1/datapath_pd*, and all of the AL scripts are in directory */pc/e/4/encoder/jiangh_test/cadabra19991/classic-generic/CLASSIC-SC/Alscripts.*

1. To Start the Datapath Compiler

It is assumed that the environment variables for Java and CLASSIC-SC have been set up correctly. To start the datapath compiler, type:

% ***java gui.ClassicDP***

A *DPCompiler* window will be displayed. This is the main window that you can use to capture cells and datapaths.

2. To Capture Cells

(1). In the *DPCompiler* window, click on **Cell**.

A *Cell Information* window appears.

(2). In the *Cell Information* window,

- To load an existing cell,

  a. Select **File** → **Load a cell**. The *Load a cell* form appears, enter the corresponding information for *Cell name* and *Drive strength*.

b. Click on **OK**.

- To save a cell,

  a. Select **File** → **Save a cell**. The Java class of the cell will be compiled and then saved. The AL netlist of the cell will be saved. Error message or the information indicates that the cell has been successfully saved will be displayed in the *Message* window, which is located at the lower part of the *Cell Information* window.

- To add a standard cell,

  a. Select **Add cell** → **Add a standard cell**.

  b. Enter the corresponding information in the following fields:

CELL Name:	type in the name of the cell
nX Drive:	type in the factor of the drive strength
Define the JAVA class of the cell:	type in the Java class of the cell
Define the CLASSIC-SC netlist of the cell:	type in the AL netlist of the cell

3. To Capture Datapaths and Generate Layouts

(1). In the *DPCompiler* window, click on **DataPath**.

A *Generate Layout for Datapath* window appears.

(2). In the *Generate Layout for Datapath* window,

- To load an existing datapath design,

  a. Select **File** → **Load**. A *Load a datapath design file* dialog window appears.

  b. Navigate the file system and click the name of the file that contains the design. Click on **Open**.

- To save a datapath design,

  a. Select **File** → **Save**. A *Save a datapath design file* dialog window appears.

  b. Navigate the file system, and click or type in the file name that will contain the design. Note that the file name must be the same as the name of the corresponding class of the datapath. Click on **Save**.

- To compile the Java class of a datapath

  a. Select **Design** → **Compile.** The error message or the information that indicates that the design has been successfully compiled will appear in the lower part of the window.

- To generate layout for a datapath,

  a. Select **Design** → **Generate Layout.** A *Set parameters* window appears.

  b. Enter all the information requested, including the name of the datapath, the width of bits, the name and the bus width for each IO net. Click on **OK**.

     ♦ If the layout is generated successfully without any interruption, the layout will be shown in the *Datapath Layout* window. You can manipulate the layout by doing the step *c*.

     ♦ If an incompletely-routed cell layout is obtained, a dialog window will be prompted showing three choices:

       ° Go to CLASSIC-SC to complete the routing manually, save the result, and notify the system to continue by clicking on **Continue**.

       ° Click on **Stretch** to notify the system to continue by stretching cells.

       ° Click on **Abort** to notify the system to stop.

     ♦ If an infeasible layout is obtained after the compaction, a dialog window will be prompted showing two choices:

       ° Click on **Stretch** to notify the system to continue by stretching cells.

- Click on **Abort** to notify the system to stop.

c. In the *Datapath Layout* window, Click right mouse button. A menu appears. You can choose *zoom in, zoom out, fit view, show all routes, show over-cell routes, show channel routes*, and *show layout*.

- To export a datapath layout,

a. Select **Design** → **Export Layout.** An *Export the datapath in GDSII* dialog window appears.

b. Navigate the file system, and click or type in the file name that will contain the GDSII layout of the datapath. Click on **Save**.

References

[Amm93] L. B. Ammar, A. Greiner, “A High Density Datapath Compiler: Mixing Random Logic with Optimized Blocks”, Proceedings of European Conference on Design Automation, pp. 194-198, 1993.

[Boo94] G. Booch, Object-oriented Analysis and Design with Applications, Benjamin/ Cummings Pub. Co., 1994.

[Cad99] Cadabra Design Libraries Inc., CLASSIC-SC Guide to Classes, 1999.

[Cad99] Cadabra Design Libraries Inc., CLASSIC-SC Reference Manual, 1999.

[Cad99] Cadabra Design Libraries Inc., AL Reference Guide, 1999.

[Cad00] Cadabra Design Libraries Inc., http://www.cadabratech.com, 2000.

[Cai90] H. Cai, S. Note, P. Six, H. De Man, “A Data Path Layout Assembler for High Performance DSP Circuits”, 27th ACM/IEEE Design Automation Conference, pp. 306-311, June 1990.

[Cha98] P. Chan, R. Lee, The Java Class Libraries, 2nd Edition, Vol. 1, Vol. 2, Addison-Wesley, 1998.

[Coh91] A. B. Cohen, M. Shechory, “Track Assignment in the Pathway Datapath Layout Assembler”, IEEE Conference on Computer-aided Design, pp. 102-105, 1991.

[Coh93] A. B. Cohen, M. Shechory, “Pathway: A Datapath Layout Assembler”, Synthesis for Control Dominated Circuits, Elsevier Science Publishers B. V. (North-Holland), pp. 119-131, 1993.

[Con90] J. Cong, C. L. Liu, “Over-the-Cell Channel Routing”, IEEE Trans. on Computer-aided Design, Vol. 9, No. 4, pp. 408-418, April 1990.

[Deu76] D. N. Deutsch, “A ‘Dogleg’ Channel Router”, 13th Design Automation Conference, pp. 425-433, June 1976.

[Fow97] M. Fowler, K. Scott, UML Distilled: Applying the Standard Object Modeling Language, Addison Wesley Longman, 1997.

[Gaj88] D. D. Gajski, Silicon Compilation, Addison-Wesley, 1988.

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[Goo99] J. Goodman, A. Chandrakasan, A. P. Dancy, “Design and Implementation of a Scalable Encryption Processor with Embedded Variable DC/DC Converter”, IEEE/ACM Design Automation Conference, pp. 855-860, June 1999.

[Gor89] R. Gordon, S. Mcneary et al., “An N-Bus Datapath compiler for IC Design”, IEEE Custom Integrated Circuits Conference, pp. 23.3.1-23.3.4, 1989.

[Gup82] U. I. Gupta, D. T. Lee, J. Y.-T. Leung, “Efficient Algorithms for Interval Graphs and Circular-Arc Graphs”, Networks, Vol. 12, pp. 459-467, 1982.

[Gup97] R. Gupta, Y. Zorian, “Introducing Core-Based System Design”, IEEE Design & Test of Computers, pp. 15-25, Oct.-Dec. 1997.

[Hai97] S. Haider, “Datapath Synthesis and Optimization for High-Performance ASICs”, On-line: http://www.synopsys.com/products/datapath/datapath_bgr.html, June 2000.

[Keu97] K. Keutzer, A. R. Newton et al., “The Future of Logic Synthesis and Physical Design in Deep-Submicron Process Geometries”, International Symposium on Physical Design, pp. 218-224, April 1997.

[Lef97] M. Lefebvre, D. Marple, C. Sechen, “The Future of Custom Cell Generation in Physical Synthesis”, 34th Design Automation Conference, pp. 446-451, June 1997.

[Len98] P. Lenne, A. GrieBing, “Practical Experiences with Standard-Cell Based Datapath Design Tools”, IEEE/ACM Design Automation Conference, pp. 396-401, 1998.

[Pre88] B. T. Preas, M. J. Lorenzetti, Physical Design Automation of VLSI Systems, Benjamin/Cummings Pub. Co., 1988.

[Rab96] J. M. Rabaey, Digital Integrated Circuits: a Design Perspective, Prentice Hall, 1996.

[Rin97] A. M. Rincon, C. Cherichetti et al., “Core Design and System-on-a-Chip Integration”, IEEE Design & Test of Computers, pp. 26-35, Oct.-Dec. 1997.

[Rum99] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual, Addison-Wesley, 1999.

[She95] N. A. Sherwani, S. Bhingarde, Routing in the Third Dimension: from VLSI Chips to MCMs, IEEE Press, 1995.

[Shi90] M. Shiochi, Y. Tanaka et al., “New Design Approach for Configurable Data-Path”, IEEE Custom Integrated Circuits Conference, pp. 14.5.1-14.5.4, 1990.

[Sun00] Sun Microsystems, http://www.java.sun.com, 2000

[Tal91] M. Taliercio, G. Foletto, L. Licciardi, “A Procedural Datapath Compiler for VLSI Full Custom Applications”, IEEE Custom Integrated Circuits Conference, pp. 22.5.1-22.5.4, 1991.

[Tsu94] Y. Tsujihashi, H. Matsumoto et al., “A High-Density Data-Path Generator with Stretchable Cells”, IEEE Journal of Solid-State Circuits, Vol. 29, No. 1, pp. 2-7, Jan. 1994.

[Usa91] K. Usami, Y. Sugeno et al., “Hierarchical Symbolic Design Methodology for Large-Scale Data Paths”, IEEE Journal of Solid-State Circuits, Vol. 26, No. 3, pp. 381-385, March 1991.