Iris: An Integrated Circuit Layout Automatic Generator

by

Yang Lu

B.S., Shandong University, 1993

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

August 2003

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Aug 28, 2003

Abstract

In integrated circuit design, layout generation is tedious, time-consuming and error-prone. Motivated by seeking an alternative to manual layout design, I implmented a CAD tool, Iris, dedicating to layout generation automation. By using Iris, the designer describes the circuit netlist and relative placement of each transistor and signal in the high level language Java. Iris works out the details of every design stage and produces the final layout. Experimental results show that Iris generates layouts which are comparable to manual layouts with much less effort by the designer.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to thank my supervisor, Dr. Mark Greenstreet, for his inspiration and guidance. I would also like to thank my project partners Lan Lin and Alvin Albrecht.

YANG LU

The University of British Columbia

May 2003

To my parents

Chapter 1

Introduction

Integrated circuit technology has evolved from the integration of a few transistors on a single chip to the integration of millions of transistors per chip. The design technology has advanced to deep sub-micron processes. Modern circuits are too complex and difficult for an individual to comprehend completely. Therefore automating the design process has become the crucial issue, and a comprehensive set of CAD tools are essential.

1.1 Motivation

VLSI chip design involves several phases starting with the specification of the system and going through with the packaging and testing of the chips. Physical layout is an extremely tedious, time-consuming and error-prone process among the phases. To make the layout problem more tractable, this phase is often divided into several steps including partitioning, placement and floorplanning, global routing, detailed routing and compaction.

Almost all the problems related to each steps of layout design are NP-complete. Considerable research has been done in this area both theoretically and practically. Theoretical research has focused primarily in the area of the design and analysis of algorithms. Practical

research focuses on the development of various CAD tools which are based on particular algorithms and address some practical issues. The present research is of the practical sort.

This thesis describes a CAD tool, Iris, which I developed to automate the layout design process. Iris addresses some issues related to the design phases including placement, global routing, detailed routing and compaction. Iris deals with channel routing problems and uses some simple algorithms to do global and detailed routing. To my knowledge, the approach that Iris uses is different from that of any existing CAD tool. Most existing CAD tools use grid-based approaches, which make it difficult to route nets with varying wire width and a large amount of memory are usually needed for dealing with the grid. Iris employs a gridless approach while using some techniques that combine both grid-based and gridless methods.

1.2 Solution

Iris generates layout through several steps. In the first step of layout synthesis, the designer writes a Java program to describe the relative placement and netlist information of the circuit. The designer also chooses the layout synthesis technology and the objective function for layout optimization. Once the circuit descriptions are given, Iris works out the details of every design stage and produces the final layout.

Based on the placement information given by the designer, Iris optimizes the placement by implementing diffusion and contact sharing in the placement stage. With the terminal sharing information, the linear programming solver produces the transistor relative position in left-to-right order for the adjacent p-transistor and n-transistor rows within the same routing channel.

Based on the placement information generated in the previous step, detailed routing or channel routing is done in every routing channel. In this stage, compaction is performed with jog

insertion. The constraints of detailed routing are solved with a linear programming solver, and a new placement is produced.

The new placement is used as the initial placement in the final step. Those nets that cover more than one routing channel are routed as well as those nets within a routing channel. More jogs are inserted and more constraints are added. The linear programming solver works out the final layout at the end of this step.

Iris has three packages. The *Circuit* package describes, places and routes the circuit. The *Constraint* package solves the constraints. The *Magic* package outputs the layout in the format used by the public domain layout editor Magic [Ousterhout 84]. The architecture of Iris is shown in figure 1.1.

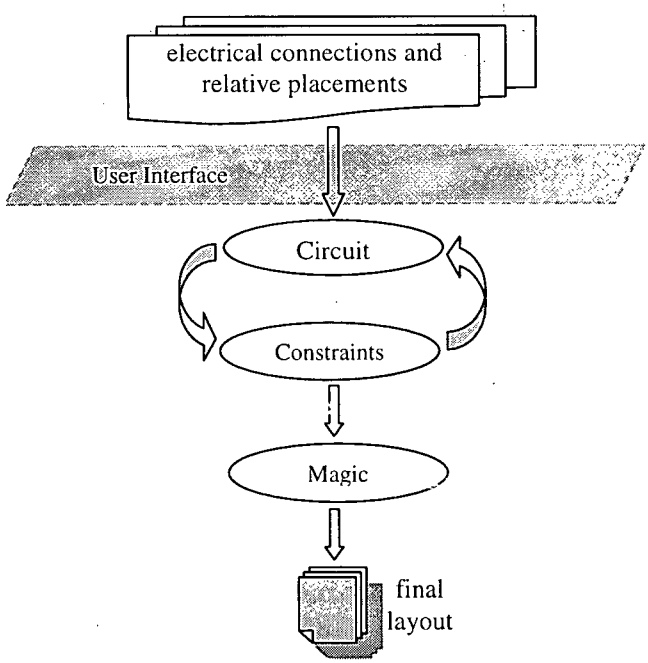


Figure 1.1 The Architecture of Iris

1.3 Contributions

The primary contribution of my work is the implementation of a CAD tool, which describes the circuit placement and netlist using a high level language, and automates the design stages which includes placement, detailed and global routing, and compaction. The layouts generated by Iris demonstrate that automatic layout design can be as area efficient as manual layout with greatly improved design productivity. Specifically, the contributions are,

- A novel, simple heuristic method for routing
- The implementation of compaction, which is integrated into the routing design stage
- Objective functions are designed and the layouts are based on linear programming
- Supporting multiple fabrication technologies and their design rule sets (i.e., MOSIS SCMOS and TSMC 0.18-micron technologies)
- A hierarchical, object-oriented user interface
- Aspect oriented programming techniques are applied in the development of Iris

1.4 Thesis Outline

This chapter gives an introduction of the problem to be addressed, the overview of Iris approach, and the main contributions. In Chapter 2, I will give a brief overview of related work to show the difference between previous approaches and the approach of Iris. In that chapter, I will also give a brief introduction of what had already been done with Iris before my work, which was explained in detail in [Lin01]. Chapter 3 presents the detailed implementation including the user interface, the routing algorithms and the constraint solving methods. Chapter 4 presents some test results. Chapter 5 gives conclusions, along with some suggestions for possible future research.

Chapter 2

Background

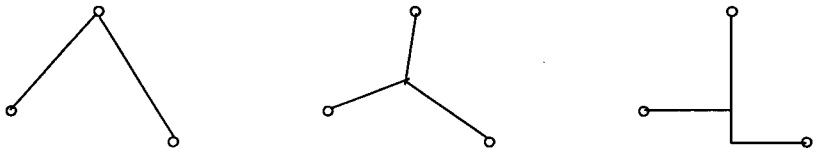
2.1 Related Work

The objective of VLSI layout design automation is to investigate the optimal arrangements and interconnections of devices and it is essentially the study of algorithms and data structures. Many different approaches have been proposed to solve the problems related to each phase of layout design.

Routing is a very important and difficult phase in layout design. Routing quality is decided by several factors. Some important factors include completing the routing of all the nets, minimizing the routing area, minimizing the wire length, minimizing the number of vias and jogs, meeting the timing constraints, minimizing the crosstalk between nets and minimizing the delay for critical nets. Due to the complexity of routing, it is usually divided into two phases, global routing and detailed routing. Global routing generates the coarse route and detailed routing generates the exact route for each net.

Global routing often uses a sequential approach choosing the shortest path for each net as it is considered. Maze routing algorithms and line-probe algorithms can be used to route two-

terminal nets. Finding the shortest path for a multi-terminal net is equivalent to finding a minimum Steiner tree. As only horizontal and vertical directions are considered in our layout, this problem becomes finding the minimum Rectilinear Steiner tree (RST). Steiner Minimum Tree (SMT) is a Steiner tree with minimum cost. SMT and many of its variants are NP-complete [Garey 77]. Since SMT is related to minimum spanning tree (MST), MST can be used to approximate SMT. The figure below shows the MST, SMT and RSMT for a three-terminal net.



Minimum Spanning Tree Steiner Minimum Tree Rectilinear Steiner Minimum Tree

Figure 2.1 MST, SMT, RSMT of Three-Terminal Net

The Kruskal [Kruskal 56] and Prim [Prim 57] algorithms are common algorithms used to solve minimum spanning tree problem. They have polynomial time complexity. [Hwang 76] showed that $\frac{\cos t_{mst}}{\cos t_{rsmt}} \leq \frac{3}{2}$. As a result, many heuristic algorithms start with MST and apply local modification to obtain an approximate RSMT.

The channel routing problem is a restriction of the detailed routing problem with the property that all terminals are on the top or bottom side of the routing region. The horizontal routing region in the middle is the channel. This matches the arrangement of transistor rows and routing rows that we assume in Iris. The problem of finding a channel route with the minimum number of routing tracks is NP-complete [Szymanski 85]. Therefore, many algorithms dealing with routing problems are heuristic in nature.

There are several criteria for classifying the routing algorithms. The algorithms can be classified as those for solving single-layer routing problem or those for solving multi-layer routing problem. The algorithms can also be classified as grid-based in which a grid is super-imposed on the routing region and the wires are restricted to follow the grid lines, or gridless in which wire placement is unrestricted. The algorithms can also be classified as using an unreserved layer model in which the net segment can be placed on any layer, or using a reserved layer model in which certain types of segments are restricted to particular layers. The algorithms can also be classified according to the heuristic they use, such as greedy routers.

One special case of the single-layer routing problem is river routing. Although Iris allows routing in three layers, river routing is still applicable in our design. Given a transistor placement, it is quite possible that the gate nets are river routable. River routability can be determined based on the algorithm described in [Hsu 83]. If the gate nets are river routable, we can route these nets in polysilicon layer only. Then we can use multi-layer routing algorithms to route the rest of the nets.

If vias are allowed, then the channel routing problem can be routed in two layers. Therefore currently Iris primarily routes in metal1 and metal2 layers, and uses polysilicon layer to connect transistor gate to the routing channel. In the future, we may consider routing in more metal layers to produce more compact design. Several basic algorithms and many of their extensions address two-layer channel routing problem.

The Left-Edge algorithm (LEA) [Hashimoto 71] sorts horizontal segments from left to right and assigns the segments to a track so that no two segments overlap. Iris scans the terminals in a left-to-right order, compares the left edge variable of the terminal and determines which track to assign. Dogleg routing [Deutsch 76] is based on LEA and allows doglegs. A dogleg is a

vertical segment that is used to connect two horizontal segments of the same net on different tracks. Iris allows doglegs, but the doglegs are limited to the top and bottom most tracks.

Common greedy routing algorithms routes the nets from left to right using a greedy strategy. The Rivest-Fiduccia greedy channel router [Rivest 82] allows a net to be placed on more than one track and have a vertical line crossing more than one horizontal segment of the same net. Iris also uses a greedy strategy but places a net on one track only at a time.

Several factors affected the design choices of Iris. Because of the large amount of storage and the effort to maintain grids and the availability of efficient linear programming solvers we got the idea of using gridless approach based on linear programming solver. The gridless approach of Iris guarantees that the routing can always complete successfully. This is not guaranteed with most grid-based routers. The basic idea of Iris is to use a greedy heuristic method which is very similar to the Left-Edge algorithm and allows doglegs. Iris uses a simple routing heuristics so that the execution time won't be too long and the code is easy to test and debug.

2.2 Iris Approach

The first prototype of Iris was built by Lan Lin. A detailed description was presented in her thesis [Lin01] including the design flow, user interface, transistor placement and relative optimization, linear programming interface and magic output interface. Most parts of her implementation are preserved in my present work with a few improvements. Specifically, section 1.2 gives the new design flow and Iris architecture; section 3.1 shows the new user interface, transistor placement and relative optimization; section 3.3 describes the new linear programming solver; and a new function was added to the magic output interface to show the critical path of the linear programming optimization problem.

The Iris user interface is written in the Java language. The interface specification adopts a hierarchical structure. Top-level cells are composed of subcells, which have smaller subcells. At the leaves of this tree structure are cells which contain only transistors and signals. All the cells in each level of the tree structure can have signals local to the cell or global signals connected to other cells. In chapter 3 a simple adder is presented as an example of how Iris generates the final layout from the user defined interface file.

The user describes the transistor placement in the user interface file. Based on the information of the relative positions of transistors and signal connections given in the user interface file, Iris optimizes the transistor placement by implementing contact sharing and diffusion sharing. Once the device merging is complete, the circuit geometry is generated with a set of rectangles and a set of constraints.

The constaints are solved with a linear programming solver. Two solving methods were introduced, LPABO solver and depth-first-traversal solver [Lin01]. LPABO solver is not used later because of the difficulty of getting a bounded solution [Lin01]. The depth-first-traversal algorithm produced satisfying results in her prototype. However, I will show in chapter 3 that the depth-first-traversal algorithm can be used only in solving transistor placement constraints, which led to my seeking an alternative linear programming solver for solving routing constraints.

Prior to constraint solving, the variables associated with rectangle vertex coordinates have symbolic values. After the constraint solving, the variables have specific real values. Iris writes the rectagles to the layout file. The layout file is in Magic format. It can be shown and manipulated in Magic.

Lan Lin's implementation produced an optimized transistor placement. My research extended this by implementing a router which I will describe in chapter 3.

Chapter 3

Implementation

This chapter describes the implementation of Iris, including both the external user interface and the internal routing and constraint solving implementation. Combined with the background description in chapter 2, this provides a full description of Iris.

3.1 User Interface

3.1.1 Example

The formula and schematics of a one-bit full adder and an n-bit ripple-carry adder have been described in Lan Lin's thesis. I will go on using this example to illustrate how Iris generates the final layout from the schematic step-by-step.

The one-bit full adder is built on two basic blocks, the carry block and the sum block. First we will examine the carry block and the sum block individually. The following figure gives the schematic of the carry block. P_0 ... P_5, N_0 ... N_5 represent the order in which the user will place the transistors from left to right in each p-transistor and n-transistor row. The user chooses the order to optimize the transistor placement so that the layout area can be minimized

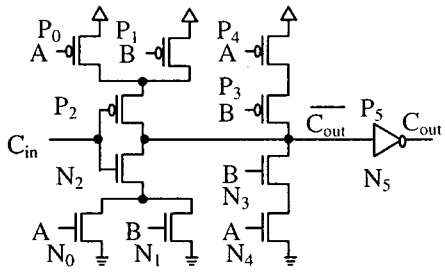


Figure 3.1 Circuit Schematic of the Carry Block

From the schematic we build the circuit layout plan as in figure 3.2.

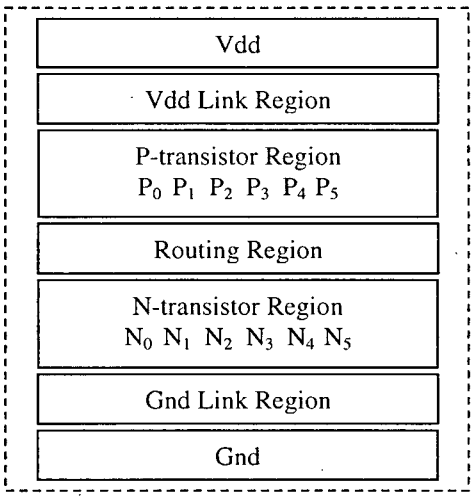
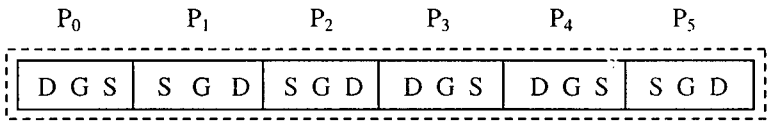


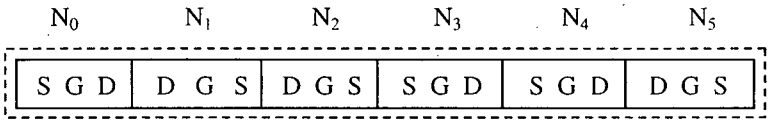
Figure 3.2 Layout Plan for the Carry Block

To implement diffusion sharing and source/drain sharing, we build the following layout plans for the p-transistor and n-transistor rows.



S: Source G: Gate D: Drain

Figure 3.3 Layout Plan for the P-Transistor Row



S: Source G: Gate D: Drain

Figure 3.4 Layout Plan for the N-Transistor Row

From the above layout plan, we write the following code for the carry block. For the complete code see appendix A.

```
public class CarryBlock {
   private DefaultSignal _Vdd;
   ...

   public CarryBlock() {
         _Vdd = new DefaultSignal();
         ...

         // Based on P-Transistor Row Layout Plan
         boolean[] pflip = new boolean[] { true, false, false, true, true, false };
         // When the value is true, we use Source-Gate-Drain order.
         // When the value is false, we use Drain-Gate-Source order.
         PTransistor[] p = new PTransistor[pflip.length];
         for (int i = 0; i < p.length; i++) {
            p[i] = new PTransistor(8);      // 8 is the transistor width in Magic's "lamda"
            if (pflip[i]) p[i] = p[i].flip();
         }
         // Based on N-Transistor Row Layout Plan
         ...

         // Netlist
         p[0].source().connect(p[1].source()).connect(p[4].source())
            .connect(p[5].source()).connect(Vdd);
         ...

         // Based on P-Transistor Row Layout Plan, place the transistors in left-to-right order
         for (int i = 0; i < p.length; i ++) {
            if (i == 0) pr = p[i];
```

```
        else pr = pr.left(p[i]);
      }
      // Based on N-Transistor Row Layout Plan
      ...

      // Based on Carry Block Circuit Layout Plan, place the circuit rows in vertical order
      c = (Vdd)
        .above(pr)
        .above(nr)
        .above(Gnd);
  }
}
```

Figure 3.5 Code Fragment from the Carry Block

Running the above code produces the layout of the carry block shown in figure 3.6.

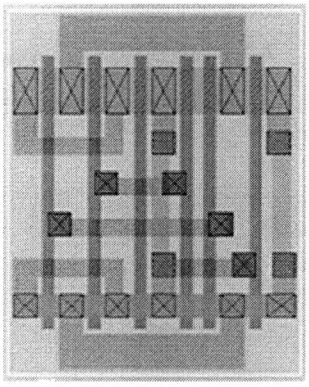


Figure 3.6 Iris Generated Layout of the Carry Block

Similarly, the schematic of the sum block is shown below.

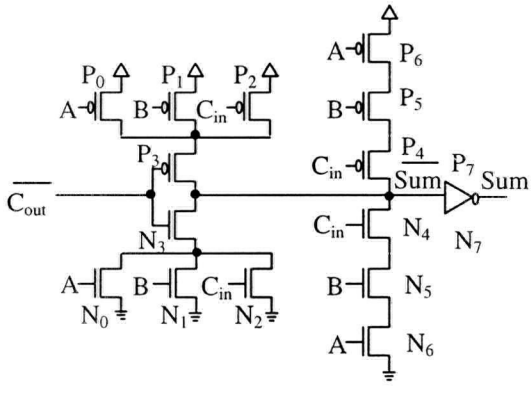


Figure 3.7 Circuit Schematic of the Sum Block

The circuit layout model, p-transistor and n-transistor row layout plan are shown below.

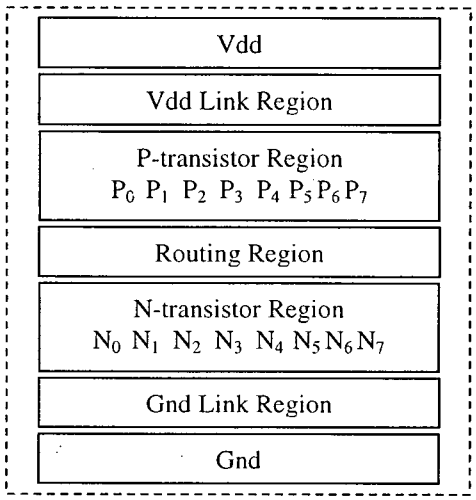


Figure 3.8 Layout Plan for the Sum Block

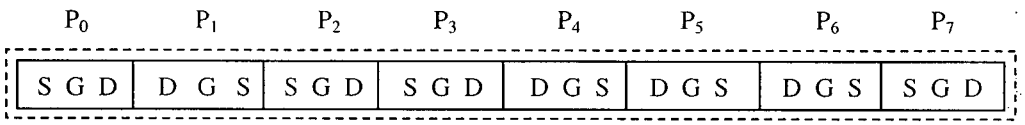


Figure 3.9 Layout Plan for the P-Transistor Row

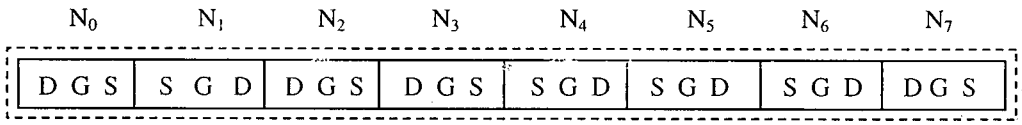


Figure 3.10 Layout Plan for the N-Transistor Row

From the above layout model, we can write the following code for the sum block. For complete code see appendix A.

```
public class SumBlock {
```

```
    private DefaultSignal _Vdd;
    ...

    public SumBlock() {
        _Vdd = new DefaultSignal();
        ...

        // Based on P-Transistor Row Layout Plan
        boolean[] pflip = new boolean[] { false, true, false, false, true, true, true, false };
        // When the value is true, we use Source-Gate-Drain order.
        // When the value is false, we use Drain-Gate-Source order.
        PTransistor[] p = new PTransistor[pflip.length];
        for (int i = 0; i < p.length; i++) {
           p[i] = new PTransistor(8);      // 8 is the transistor width in Magic's "lamda"
           if (pflip[i]) p[i] = p[i].flip();
        }
        // Based on N-Transistor Row Layout Plan
        ...

        // Netlist
        p[0].source().connect(p[1].source()).connect(p[2].source())
           .connect(p[6].source()).connect(p[7].source()).connect(Vdd);
        ...

        // Based on P-Transistor Row Layout Plan, place the transistors in left-to-right order
        for (int i = 0; i < p.length; i ++) {
           if (i == 0) pr = p[i];
           else pr = pr.left(p[i]);
        }
        // Based on N-Transistor Row Layout Plan
        ...

        // Based on Sum Block Circuit Layout Plan, place the circuit rows in vertical order
        c =  (Vdd)
           .above(pr)
           .above(nr)
           .above(Gnd);
    }
}
```

Figure 3.11 Code Fragment from the Sum Block

Running the above code produces the layout of the sum block shown in figure 3.12.

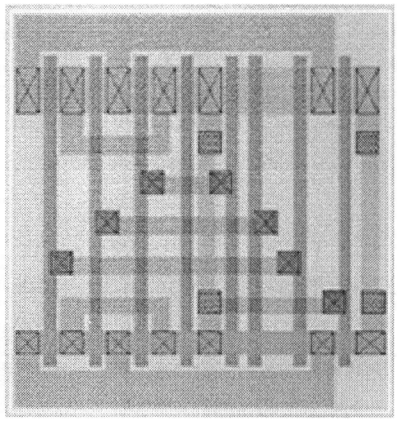


Figure 3.12 Iris Generated Layout of the Sum Block

The one-bit full adder is built from the above two blocks, the carry block and the sum block. We may put the carry block and the sum block side by side either horizontally or vertically. Here I put the carry block on top of the sum block.

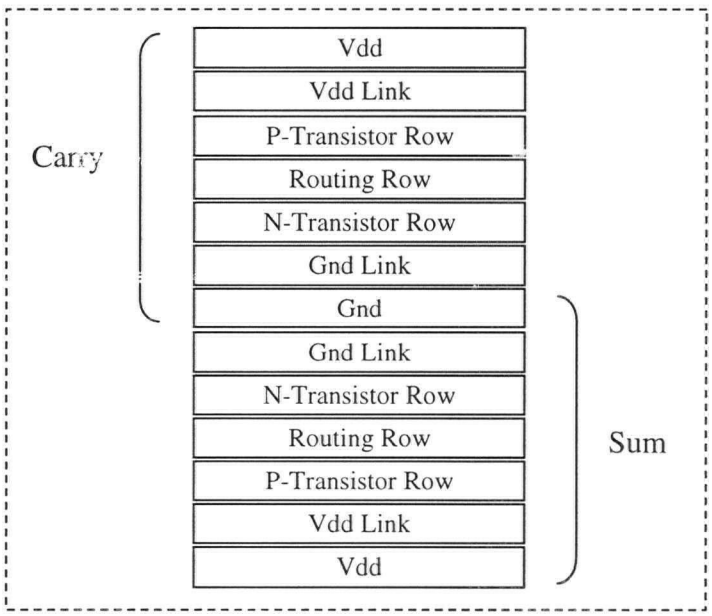


Figure 3.13 Layout Plan for the One-Bit Full Adder

By putting the carry block on top of the sum block, the sum block is flipped vertically so that the two blocks share a common ground. This flip is performed automatically by Iris. The input and output signals, such as A, B, C_{in}, C_{out}, which are local signals inside each block originally, become global signals spanning the two routing rows now.

```
public class OneBitFA {
   private DefaultSignal _Vdd;
   ...

   public OneBitFA() {
         _Vdd = new DefaultSignal();
         ...

         // build the Carry Block and the Sum Block individually
         SumBlock sb = new SumBlock();
         CarryBlock cb = new CarryBlock();

         // make signal conections between the two blocks
         sb.Vdd().connect(cb.Vdd()).connect(Vdd);
         sb.Gnd().connect(cb.Gnd()).connect(Gnd);
         A = sb.A().connect(cb.A());
         B = sb.B().connect(cb.B());
         Cin = sb.Cin().connect(cb.Cin());
         Not_Cout = sb.Not_Cout().connect(cb.Not_Cout());

         // the output signals of the one-bit full adder
         Cout = cb.Cout();
         Sum = sb.Sum();
         c = cb.circuit().above(sb.circuit());
   }
}
```

Figure 3.14 Code Fragment from the One-Bit Full Adder

Running the above code produces the layout of the one-bit full adder shown in figure 3.15. The complete code is shown in appendix A.

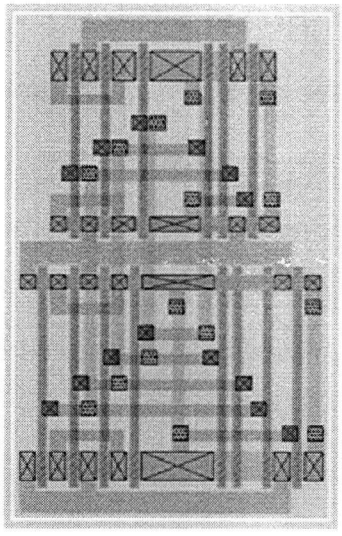


Figure 3.15 Iris Generated Layout of the One-Bit Full Adder

The n-bit ripple-carry adder is built from n copies of the one-bit full adder. We may put those one-bit full adders side by side either horizontally or vertically. Here I put them horizontally.

Figure 3.16 Layout Plan for the n-Bit Ripple-Carry Adder

By arranging the one-bit full adder cells horizontally Iris automatically combines corresponding rows of the same type. Power and ground rows are merged into rows that span all of the one-bit full adder cells. The p-transistor and n-transistor rows are concatenated with other p-transistor and n-transistor rows respectively. The routing row, power link and ground link span the whole channel. The input and output signals, such as C_{in} and C_{out}, which are local signals inside each one-bit full adder cell originally, become global signals connecting two one-bit full adder cells now.

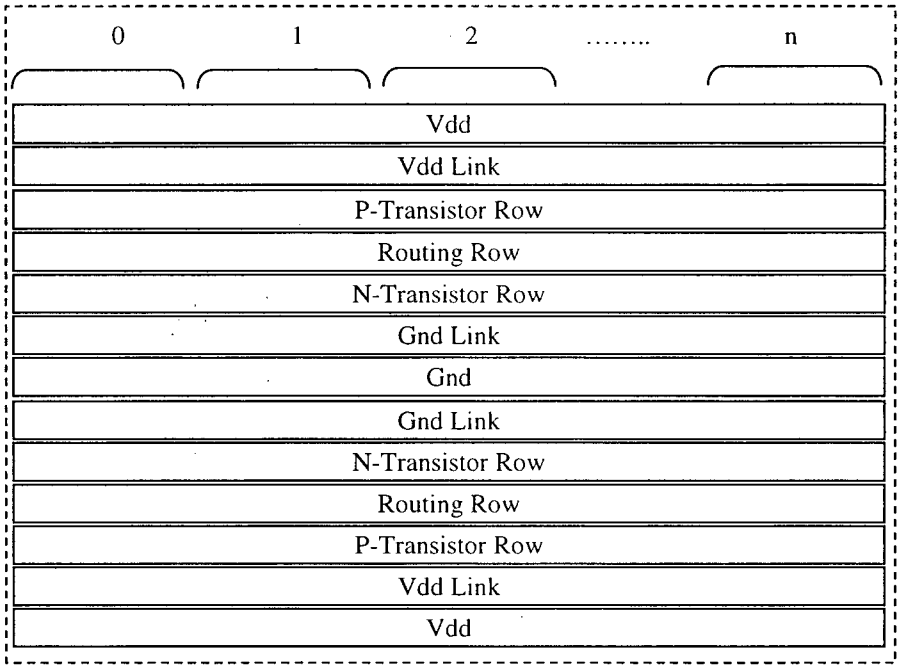


Figure 3.17 Layout Plan for the n-Bit Ripple-Carry Adder

```
public class nBitRCAdder {
   private DefaultSignal _Vdd;  public Signal Vdd() { return(_Vdd); }
   ...

   public nBitRCAdder(int n) {
         _Vdd = new DefaultSignal();
         ...
         OneBitFA fa = new OneBitFA();
         c = fa.circuit();
         A[0] = fa.A();
         B[0] = fa.B();
         Sum[0] = fa.Sum();
         Cin = fa.Cin();
         Cout = fa.Cout();
         fa.Vdd().connect(Vdd);
         fa.Gnd().connect(Gnd);

         // build one-bit full adder repeatedly
         for (int i = 1; i < n; i ++) {
            fa = new OneBitFA();
```

```
            fa.Cin().connect(Cout);
            fa.Vdd().connect(Vdd);
            fa.Gnd().connect(Gnd);
            c = c.left(fa.circuit());
            A[i] = fa.A();
            B[i] = fa.B();
            Sum[i] = fa.Sum();
            Cout = fa.Cout();
        }
    }

    public static void main(String[] args) throws Exception {
        Technology.setTech(Scmos.tech());
        int solver = GeomCircuit.Matlab;
        ObjFunc[] obj = new ObjFunc[1];
        obj[0] = new ObjFunc();
        obj[0].setWeight(ObjFunc.boundsWt, 0);
        obj[0].setWeight(ObjFunc.layerWt, 5);
        obj[0].setWeight(ObjFunc.netWt, 0);

        (new GeomCircuit((new nBitRCAdder(4)).circuit(), Technology.tech(), "nBitRCAdder",
solver, obj)).toGeomCircuit();
    }
}
```

Figure 3.18 Code Fragment from the n-Bit Ripple-Carry Adder

Running the above code produces the layout of the n-bit ripple-carry adder shown in figure 3.19. For the complete code see appendix A.

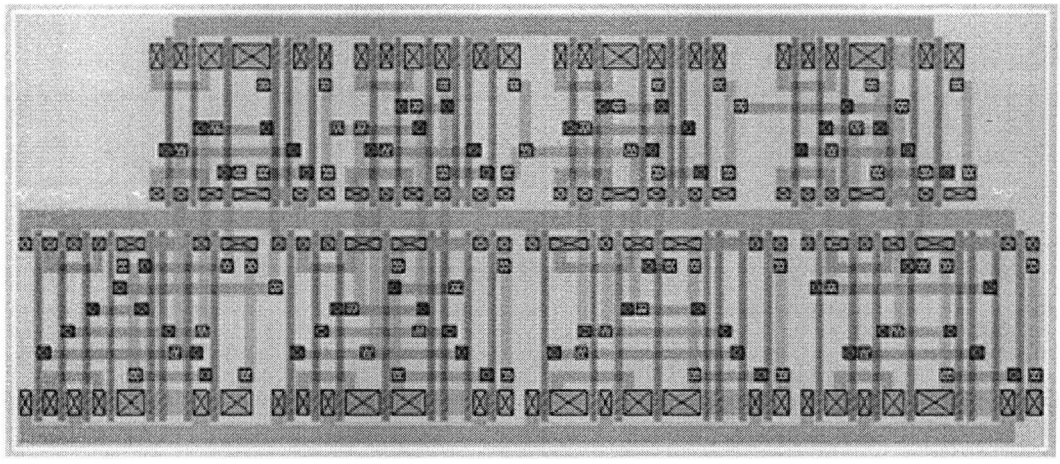


Figure 3.19 Iris Generated Layout of the n-Bit Ripple-Carry Adder

3.1.2 Class Summary

The user interface is written in the Java language. AspectJ is used in the Circuit package. The user doesn't need to be familiar with AspectJ. All the user needs to know about AspectJ is using "ajc" to compile and "ajava" to run on Solaris and Linux machines, and "ajc" to compile and "java" to run on Win2K platforms. Several templates have been given in the adder example. The user may define the cells hierarchically. The java interface files for the leaf cells are similar to the files of the carry block and the sum block. Those for the upper level cells are similar to the one-bit full adder and the n-bit ripple-carry adder. Table 3.1 gives the information that needs to be included in the user interface file.

Leaf cell	Upper level cell
1. Input/output signals, power and ground 2. (Optional) Terminal order (Source-Gate-Drain or Drain-Gate-Source) of each transistor. The default order is Source-Gate-Drain. 3. Netlist 4. The relative ordering of the transistors in the horizontal direction 5. Circuit rows (transistor rows and power/ground rows), and the relative position of these rows in the vertical direction	1. Input/output signals, power and ground 2. Lower level cells 3. Signal connections of lower level cells (Netlist) 4. The relative position of lower level cells in horizontal or vertical directions

Table 3.1 User Interface File

The signal interface and several class implementations of Signal interface are used for defining input/output signals and power/ground. Usually only Signal, DefaultSignal, Power and Ground are used in the interface file. The method Connect is used to define netlists.

Interface: Signal			
Class: DefaultSignal	Class: Power	Class: Ground	Class: Terminal

Table 3.2 Signal Interface and Class Implementations

As the circuit is defined in a hierarchical structure with upper level cells composing of lower level cells and leaf cells composing of transistors, so are the class structures. Figure 3.20 shows the class hierarchy of the circuit.

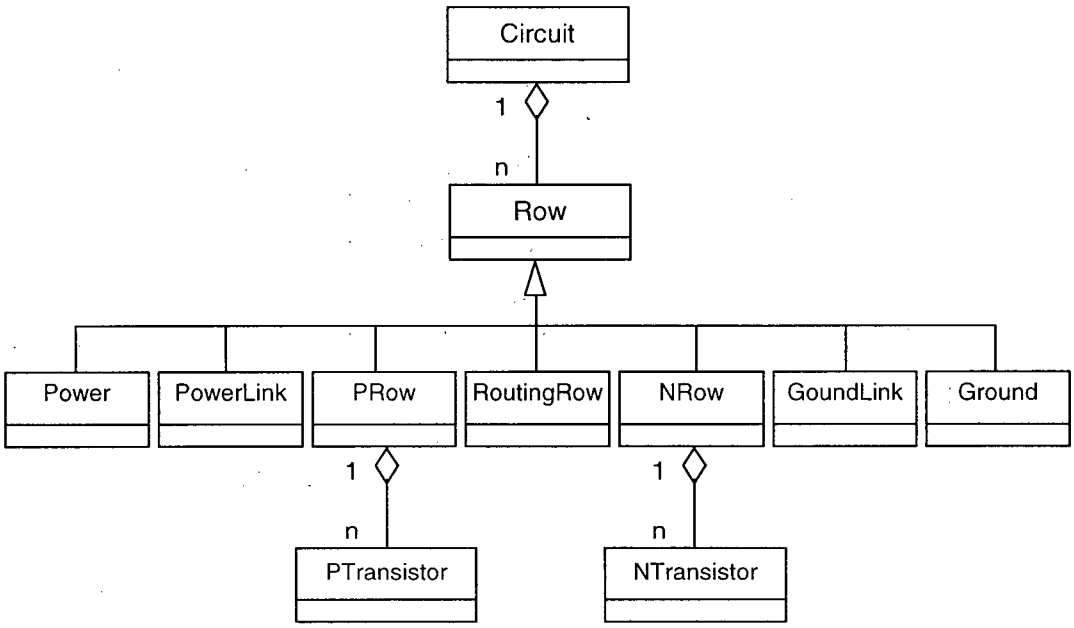


Figure 3.20 Class Hierarchy of Circuit

In this class hierarchy, only Circuit, Power, Ground, PRow, NRow, PTransistor and NTransistor are used in the interface file, which correspond to the whole circuit cell, vdd, gnd, p-transistor row and n-transistor row respectively in the actual circuit as shown in the circuit layout plan in the adder example. PowerLink, GroundLink and RoutingRow are added to the circuit automatically in the internal implementation, which correspond to vdd link, gnd link and routing rows in the actual circuit as shown in the circuit layout plan in the adder example. PowerLink is inserted between the power and p-transistor rows in the circuit. GroundLink is inserted between the ground and n-transistor rows. RoutingRow is inserted between p-transistor and n-transistor rows. Once the complete circuit is built, placement, routing and compaction are done on this complete circuit. Table 3.3 shows these classes and their methods relating to the user interface file.

Class	Method	Purpose
PTransistor/NTransistor	source() gate() drain()	Get the terminals of a transistor so that netlist can be defined on these terminals
	flip()	Define terminal order (Source-Gate-Drain or Drain-Gate-Source)
PRow/NRow	left()	Define transistor relative position in horizontal direction
Power/Ground	connect()	Define netlist
Circuit	left() above() mirrorX() mirrorY()	Define the relative position of circuit rows and cells in horizontal or vertical direction

Table 3.3 User Interface Class and Method

At the highest level interface file, a main function gives other information that Iris requires to produce an actual layout.

1. Choose technology (Currently *Scmos* or *Cmosp18*)
2. Define the objective function (Minimize the perimeter, wire-length, etc.)
3. Choose a linear programming solver (*Matlab*, *BellmanFord* or *DepthFirstSearch*)
4. Call the method to run the program

   For the example of the one-bit full adder, this calling will look like this,

   (new GeomCircuit((new OneBitFA()).circuit(), Technology.tech(), "OneBitFA", solver, obj)).toGeomCircuit();

Table 3.4 Main Function

Iris currently provides two technology files, *Scmos.java* and *Cmosp18.java*, which define the layers, the minimum width of each layer, the minimum separation of two layers so

that in the internal implementation necessary constraints are added to generate design rule correct layout. By writing a new file similar to these two files, other technologies can be used to generate design rule correct layout.

3.2 Routing

Routing take place in three phases. Each phase generates an ordered list of transistor terminals. Based on the initial placement generated in the previous phase and the user-provided information in the user interface file, the current phase generates an improved transistor placement which is used in the next phase as the initial placement until in the last phase a final layout is generated.

Phase 1: Describe the circuit only in terms of transistor rows. Add necessary horizontal constraints to guarantee that transistor rows satisfy the design rules. Solve the constraints and get the relative position of each transistor.

Phase 2: Describe the whole circuit, including transistor rows, power and ground rows, power link, ground link and routing rows. Do channel routing within each channel. Only connections within each channel are considered at this point. Global routing, i.e., connections between channels, are handled in the next and final phase. Add necessary constraints. Solve the constraints and get the relative position of each transistor.

Phase 3: Describe the whole circuit, including transistor rows, power and ground rows, power link, ground link and routing rows. Do global routing and detailed routing. Add necessary constraints. Solve the constraints and get the final layout.

For example, the FifoStage circuit has four transistor rows. From bottom to top, they are p-transistor row, n-transistor row, p-transistor row and n-transistor row. After phase 1, we get the transistor placement layout in the following figure. Phase 1 generates the ordered list of all transistors and their terminals, which will be used in phase 2 as the initial placement.

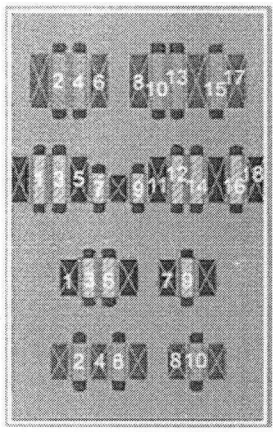


Figure 3.21 Iris Generated Layout in Phase 1

The layout described in file FifoStage1.mag only considers horizontal constraints, because vertical constraints are not important in this phase. To make the layout easy to read and design rule correct, I added vertical constraints when generating figure 3.21. I used the index number to indicate the left-to-right order at the end of the phase. Those terminals connecting to power and ground don't have the index number on them because the routing of power and ground nets is done in power link and ground link regions. Only the nets routed in a single routing row region have the index numbers. Horizontal compaction is done in this phase including diffusion sharing and source/drain sharing. This compaction generates a more accurate relative position of each terminal.

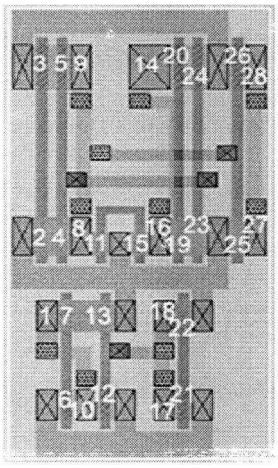


Figure 3.22 Iris Generated Layout in Phase 2

Figure 3.22 shows the layout in file FifoStage2.mag, which is generated at the end of phase 2. This layout includes the ordered list of all transistors which will be used during phase 3 as the initial placement. We can see from this layout that only detailed routing or channel routing is done in this phase. The reason that I choose to implement channel routing first and use this layout as the initial placement during phase 3 is based on the assumption that most routing is within each channel, so the local routing within each channel will have the greatest impact on the relative positions of transistors. Horizontal compaction of local routing is done in this phase, which generates a more accurate relative position of each terminal.

In chapter 4, figure 4.1 shows the final layout in file FifoStage3.mag generated at the end of the global routing process. In phase 3, I first calculate global routing information for those terminals whose nets are global nets. For each global net in each routing row, I determine at which terminal I should do global routing. Once the global routing information is achieved, both local routing and global routing are done simultaneously in phase 3.

Phase 2 and phase 3 use the same algorithm for local routing. The only difference between phase 2 and phase 3 is that phase 2 doesn't take global routing into consideration. In phase 3, Iris performs global routing first and then detailed routing, which is a common approach. Iris doesn't have a separate compaction phase. Compaction is done throughout phase 3, mixed together with routing.

3.2.1 Global Routing

Global routing determines the approximate course of connecting wires for global nets. In the case of channel routing, global nets are those nets spanning more than one routing row. Iris uses sequential routing in which one net is routed each time and the shortest path is chosen for this net. Because of the characteristics of the global routing algorithm that Iris uses, the routing order of

the global nets doesn't matter. Each time a net is arbitrarily chosen to route until all nets are routed.

The problem of finding the shortest set of connections for an n-pin net is a Steiner tree problem. Minimum spanning trees can be used to approximate Steiner trees. Prim and Kruskal algorithms for solving minimum spanning tree problem have complexity of $O(n^2)$, where n is the number of pins. Iris uses a different approach from the minimum spanning tree. The global routing algorithm Iris uses can generate good layout for most cases with better performance than Prim's and Kruskal's algorithms. The algorithm is divided into two steps. The first step is based on the sorting of routing rows. The complexity is O(nlogn) for the sort plus an additional O(n) phase following the sort, resulting in overall complexity of O(nlogn) for this step, where n is the number of routing rows. The second step is based on the sorting of the index of terminals within each routing row. The complexity is O(nlogn) for the sort plus an additional O(n) phase following the sort, resulting in overall complexity of O(nlogn) for this step, where n is the number of terminals within this routing row. The two steps are in a sequential order. The complexity of this global routing algorithm is then O(nlogn), where n is the number of terminals.

Algorithm 3.1 Global Routing

1. For each global net Do
2. Sort the routing rows in which the global net has terminals in ascending order in the *y* direction
3. For each routing row in the sorted list Do
4. Pair each routing row with its predecessor in the sorted list to get a connection
5. End For
   // Now we have a connection list for each net including all the connections generated in // the above step.
6. For each connection of this net Do
7. Sort the terminals in the bottom routing row in left-to-right ascending order
8. Sort the terminals in the top routing row in left-to-right ascending order

```
        // The intermediate rows don't matter.
9.      Denote the leftmost terminal in the bottom routing row as ch0Left
10.     Denote the leftmost terminal in the top routing row as ch1Left
        // Every terminal has an index which is produced in the previous step and
        // indicates the left-to-right order
11.     If (ch0Left > ch1Left)
12.         Find the rightmost terminal in the top routing row which has an index
            smaller than ch0Left, and denote it as ch1Routing. Do global routing at
            terminal ch0Left in the bottom routing row and at terminal ch1Routing
            in the top routing row
13.     Else // ch0Left < ch1Left
14.         Find the rightmost terminal in the bottom routing row which has an
            index smaller than ch1Left, and denote it as ch0Routing. Do global
            routing at terminal ch0Routing in the bottom routing row and at terminal
            ch1Left in the top routing row
15.     End If
16.   End For
17. End For
```

Iris tries to build a global route for one global net each time in an arbitrary order. A netlist has been built when the user interface file is given by the user. In Line 1, the algorithm chooses a global net from this netlist. Lines 2 to 5 are the first step of global routing. Every routing row has a unique id associated with it, which is called channelId. First, we find all the routing rows in which this net has terminals. Then, these routing rows are sorted in ascending or descending order by their channelId. Here I use ascending order. Next, a pair is formed between the adjacent routing rows. This step is illustrated in the following figure.

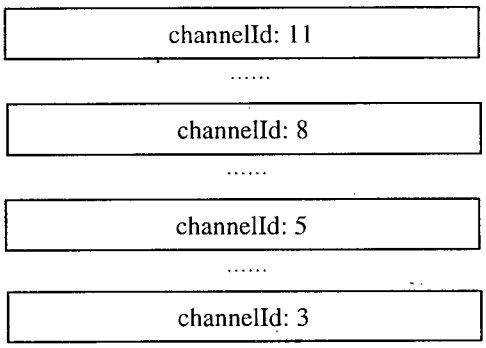


Figure 3.23 The First Step of Global Routing

Lines 6 to 16 are the second step of global routing. In this step, the algorithm tries to build one and only one connection for each pair. It determines where the connection or wire should be put. First, the terminals are sorted by their index generated in Phase 2 of routing process in left-to-right ascending order within each routing row. Then, the leftmost terminals in the routing rows of a pair are compared with each other. The bigger one will be chosen to do global routing in its routing row for this net. For the other routing row that routing has not been decided yet, a search for the terminal that has an index smaller than the one we just chose is performed. The first terminal found in the above search will be used for global routing in its routing row for this net. This step is illustrated in the following figure.

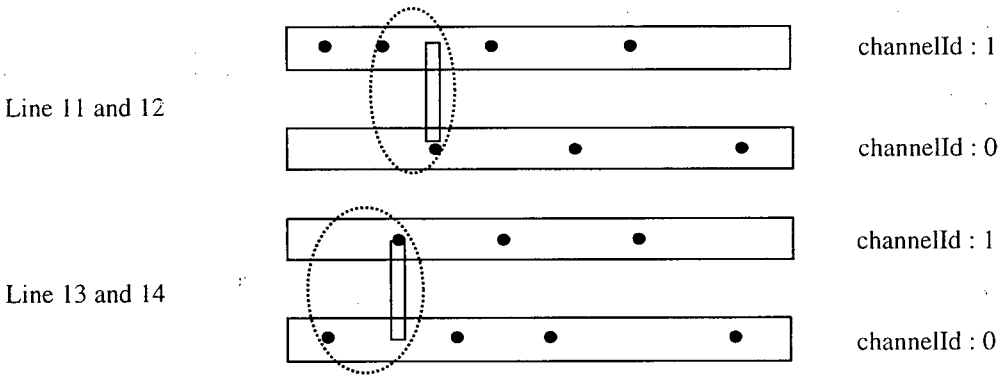


Figure 3.24 The Second Step of Global Routing

At the end of step 2, we may build the global connections for a net looking like below.

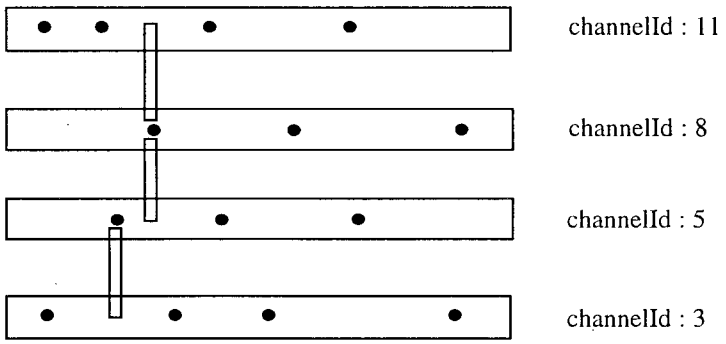


Figure 3.25 Global Connections

The above method always finds the terminals to be connected within a channel from the left side to the right side, which may cause congestion if these terminals of several global nets happen to be close to each other within a channel.

3.2.2 Detailed Routing

Detailed routing determines the exact course of wires that comply with the global routes generated in global routing phase and that obey a set of constraints. In this phase, Iris deals with the channel routing problem. Iris uses a gridless approach in that the detailed routing algorithm does not rely on the grid to determine the course of wires but relies solely on the constraints.

Iris applies a greedy method to solve the channel routing problem. The greedy method attempts to construct an optimal solution in stages. At each stage it makes a decision that appears to be the best at the time. A decision made in one stage will not be changed in a later stage. Iris handles one terminal in one stage. Several operations are performed for a terminal in one stage. In the worst case an operation has complexity of O(n). The complexity of the whole detailed routing algorithm is $O(n^2)$, where n is the number of terminals.

Iris performs detailed routing on three layers, polysilicon, metal1 and metal2. All three layers can be used to route within a channel. Only metal2 is used to route between channels. There is no limitation for the directions of the three layers. They can be placed in both horizontal and vertical directions, although polysilicon and metal2 are primarily used for vertical routing and metal1 is primarily used for horizontal routing.

Detailed routing is performed in two regions as shown in the layout plan of the adder example, power/ground link regions and routing row regions. For the power/ground link routing we can simply add a metal1 rectangle to connect the source or drain terminal with the power or ground wire when the terminal is in the same net as power or ground. For the routing row region a two-step routing procedure is performed. The detailed routing flow is shown below.

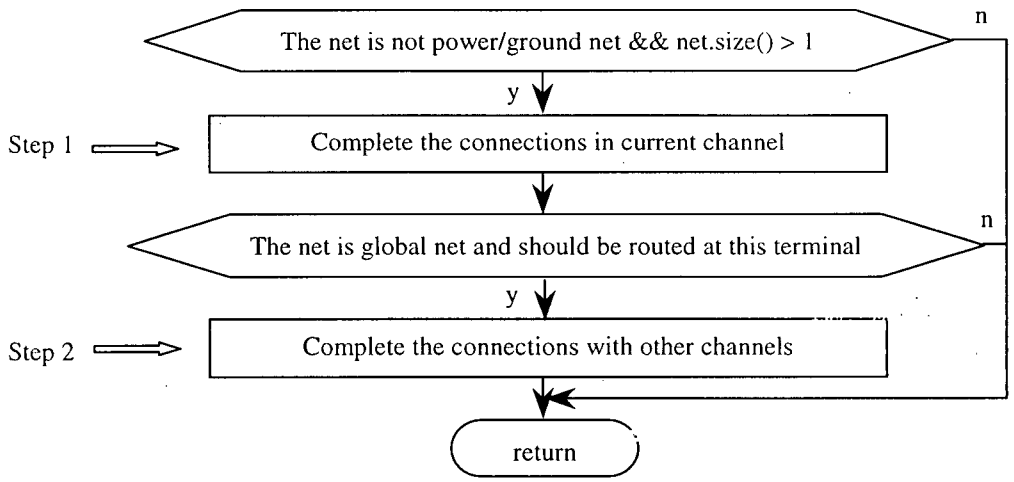


Figure 3.26 Detailed Routing Flow

The basic idea of Iris channel router is to route each terminal in the left-to-right order generated in Phase 2 of routing process, check which track the net is using and which track should be used for later routing if this is not the last terminal of this net, add necessary connections for this terminal and the tracks, and necessary constraints to satisfy the design rules.

One net can switch to another track even if it is using a track. The switching strategy is to make use of the top most and bottom most tracks most, and release the tracks in the middle. The reason for doing this is that the top most and bottom most tracks can not be released in most situations because they are used for the vias which are necessary for routing the source or drain terminal. Figure 3.27 shows an example illustrating why switching to another track is necessary for efficient area utilization.

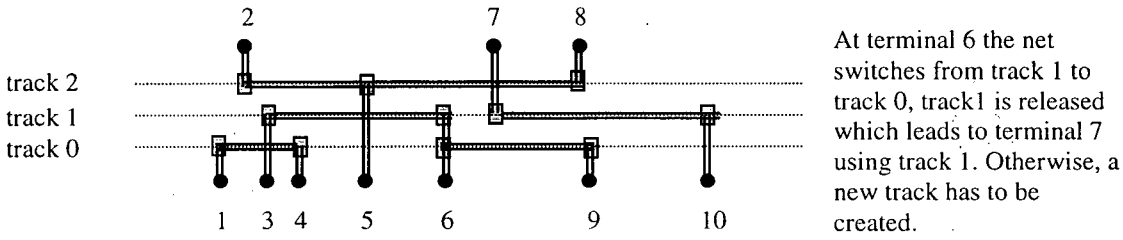


Figure 3.27 Track Switching for Efficient Area Use

Deciding when to switch and which track to switch to depends on whether the top most and bottom most tracks are currently used by other nets and whether the current net has terminals not yet routed in the current channel and other channels. If the current net has terminals not yet routed in the current channel, I denote *localFutureUse* as true. If the current net needs to do global routing at this terminal, I denote *globalFutureUse* as true. Next, I determine whether the top most and bottom most tracks are available for local routing and global routing. This procedure is similar for both local routing and global routing. First I check whether the bottom most and top most tracks are currently used by other nets. If they are used by other nets, then they are not available for the current net. Next I check those terminals located between the current terminal and the next terminal belonging to the current net in current channel to see whether they are source/drain or not. If they are source or drain, and they are not power or ground net, then the top most track will not be available if that terminal is at the top of the

channel, and the bottom most track will not be available if that terminal is at the bottom of the channel. If only the bottom most track is available, I denote *availGlobal* or *availLocal* as 0. If only the top most track is available, I denote *availGlobal* or *availLocal* as 1. If both tracks are available, I denote *availGlobal* and *availLocal* as 2. If *availLocal* and *availGlobal* conflict, that is, *availLocal = 0* and *availGlobal = 1*, or *availLocal = 1* and *availGlobal = 0*, then neither track can be used for the current net. If both tracks are available, I choose the track closest to the current terminal with the exception shown in figure 3.28. If neither track is available, I find an empty track in the middle tracks. If no empty track is available, a new track is created.

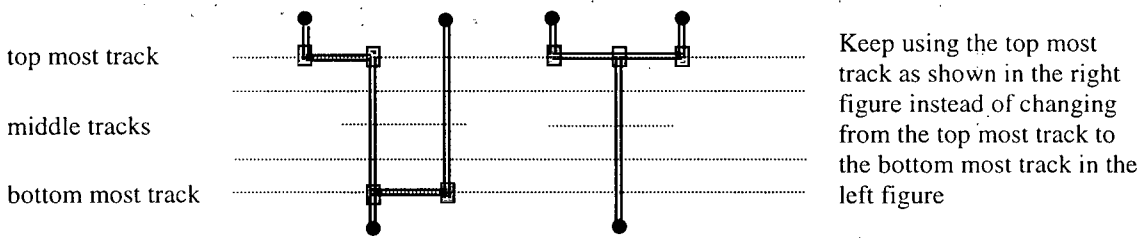


Figure 3.28 Choosing of Bottom Most Track and Top Most Track

Determining which track is being used and which track will be used by the current net is performed in step1 of detailed routing shown in figure 3.26. At the end of this process, I have *usingTrack* and *toUseTrack* being either true or false depending on whether the current net is using a track or not and whether the current net will use a track or not. I also have *usingTrackIndex* and *toUseTrackIndex* being an integer number to show the track index. For different combination of *usingTrack* and *toUseTrack*, different procedure shown in the following table is performed to complete step 1 of detailed routing.

UsingTrack	toUseTrack	usingTrackIndex = toUseTrackIndex	Procedure
False	true	X	StartUseTrack
True	false	X	EndUseTrack
True	true	Yes	UseSameTrack
True	true	No	UseDiffTrack

Table 3.5 Step 1 Procedure

Throughout step 1 compaction is done through via elimination by either vertical alignment or jog insertion. Vertical alignment can be applied to polysilicon when the terminals on both sides of the channel are gate and they belong to the same net, and can be applied to metal2 when the terminals on both sides of the channel are source or drain and they belong to the same net. Vertical alignment eliminates unnecessary polycontacts in the case of polysilicon and vias in the case of metal2. Jogging adds horizontal polysilicon or metal2 when otherwise only metal1 can be placed horizontally, and eliminates unnecessary polycontacts or vias. Jogging can be done in the following two situations shown in figure 3.29.

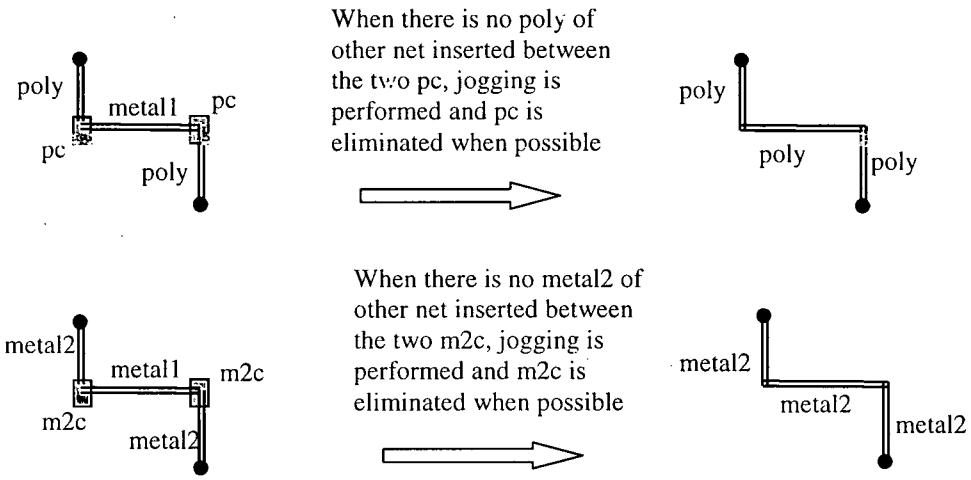


Figure 3.29 Two Jogging Situations

Once vertical alignment and jog insertion are determined, the procedure StartUseTrack, EndUseTrack, UseSameTrack and UseDiffTrack are easy to write. In the pseudocode below, whenever a rectangle is added, relative constraints are also added.

```
StartUseTrack
    If current terminal is Gate
        Extend the polysilicon to the track
        Add polycontact on this track
    Else // current terminal is Source or Drain
        Add vertical metal2 to connect the terminal to the track
        Add m2contact on the track
    End If
End StartUseTrack
EndUseTrack
    If current terminal is Gate
        Extend the polysilicon to the track
        If jogging
            Add horizontal polysilicon on the track
        Else
            Add horizontal metal1 on the track
            Add polycontact to connect vertical polysilicon and horizontal metal1
        End If
    Else // current terminal is Source or Drain
        Add vertical metal2 to connect the terminal to the track
        If jogging
            Add horizontal metal2 on the track
        Else
            Add horizontal metal1 on the track
            Add m2contact to connect vertical metal2 and horizontal metal1
        End If
    End If
End EndUseTrack
```

UseSameTrack is similar to EndUseTrack except that a polycontact is added for later routing on the track when the terminal is gate, and an m2contact is added for later routing on the track when the terminal is source or drain.

UseDiffTrack is similar to UseSameTrack except that the vertical polysilicon extended from the terminal needs to connect both the *usingTrack* and *toUseTrack* when the terminal is gate, and the vertical metal2 extended from the terminal needs to connect both the *usingTrack* and *toUseTrack* when the terminal is source or drain.

Step 2 of detailed routing connects the current channel with the other channel when the current net is a global net. The track used for this connection is determined in step 1. Compaction is done through via elimination by either vertical alignment or jog insertion which is same as in step 1. The pseudocode below shows this procedure.

```
Step 2 of Detailed Routing
    // For the current channel, when current terminal is Source or Drain, we do vertical
    // alignment, add nothing
    If current terminal of current channel is Gate
        Add horizontal metal1 and m2c on the track of the current channel
    End If
    If the last terminal of the other channel is Gate
        Add horizontal metal1 and m2c on the track of the other channel
    Else // Source or Drain
        If Jogging
            Add horizontal metal2 on the track of the other channel
        Else
            Add horizontal metal1 and m2c on the track of the other channel
        End If
    End If
    Add vertical metal2 to connect the two m2c
End Step 2 of Detailed Routing
```

The following figure shows the four situations that Step 2 of detailed routing deals with.

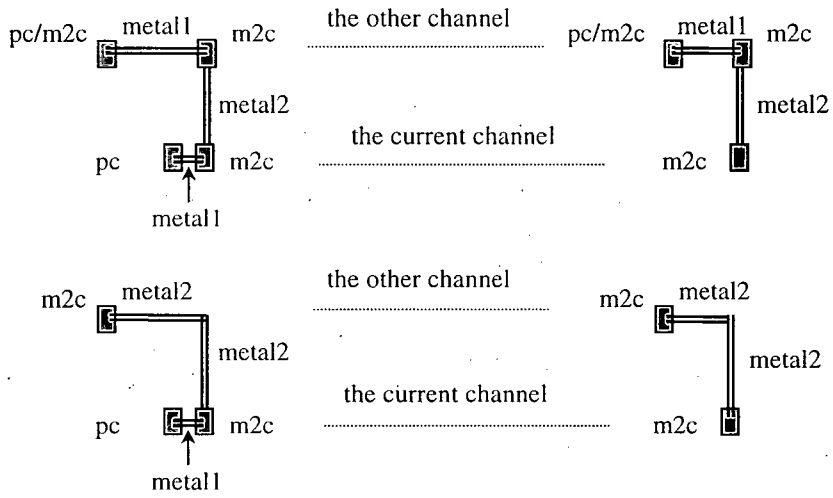


Figure 3.30 Step 2 of Detailed Routing

Before jogging and other optimization, I got the layout with area ratio to the hand layout 1.8:1 without doing global routing. From the test result in chapter 4, we can see that the optimization significantly improved the layout area with the result area to the hand layout area ratio of 0.99:1.

3.3 Constraint Solving

The integrated circuit layout has to satisfy a set of design rules including size rules, separation rules and overlap rules to guarantee correct fabrication. The geometry of the circuit layout is represented in terms of rectangles. The design rules are then turned into a set of constraints over the four variables indicating the left, bottom, right and top sides of the rectangle, and the layout problem then becomes a linear programming problem.

Two solving methods for linear programs were considered in Lan Lin's thesis. The LPABO [Park 00] solver was first tried but rejected later because of the difficulties of constructing a cost function. A depth-first-traversal algorithm was used in her prototype

successfully and satisfying results were obtained. In this thesis, I describe two other solving methods. There are two reasons that lead to my seeking alternative solving methods. First, the depth-first-traversal algorithm doesn't take into consideration the optimization cost function. Second, the depth-first-traversal algorithm requires that no cycle exists in the constraint graph. It doesn't work when constraints of the form $v_0 - v_1 \geq a$ and $v_1 - v_0 \geq b$ exist, which are produced during the reduction of constraints generated from routing. The following figure shows an example where such constraints occur.

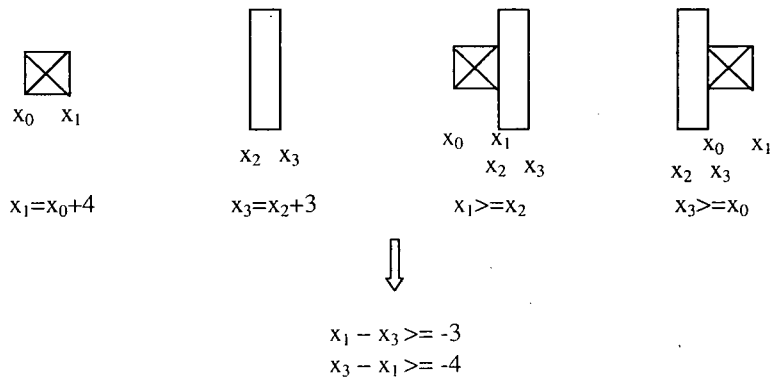


Figure 3.31 Via slides on Metal2

Although the depth-first-traversal algorithm has the two drawbacks mentioned above, it has one important good thing: its running time is fast. The time complexity is O(n+m) with n unknowns and m constraints. It can be used in phase 1 of routing, because phase 1 of routing has the property that no cycle exists in the constraint graph. This is due to the fact that in phase 1 all rectangles are put side by side in horizontal direction in each transistor row. These rectangles are either adjacent to each other or separated from each other. No rectangles can slide on other rectangles. If we want to add extra optimization features to phase 1, we still need the linear programming solvers which I will describe below.

3.3.1 Matlab Linear Programming Solver

Matlab has an optimization toolbox that is capable of solving linear programming problems. The linear programming problem is expressed in the following form.

$$\min_x f^T x \quad \text{such that} \quad \begin{aligned} A \cdot x &\le b \\ Aeq \cdot x &= beq \\ lb &\le x \le ub \end{aligned}$$

where *f*, *x*, *b*, *beq*, *lb*, and *ub* are vectors and *A* and *Aeq* are matrices, and vector inequalities are evaluated elementwise.

The function *linprog* can be used to solve this kind of problem. It has several forms which have different numbers of input and output arguments. Iris uses the following form.

$$[x, fval, exitflag, output] = linprog(f, A, b, Aeq, beq, lb, ub)$$

lb and *ub* defines a set of lower and upper bounds on the variables, *x*, so that the solution is always in the range $lb \le x \le ub$. The output argument *fval* represents the value of the objective function *f'x*. A positive value of *exitflag* indicates that *linprog* converged successfully. The argument *output* provides other information including the algorithm that *linprog* used to solve the linear programming problem.

Matlab has two options for solving linear programming problem using *linprog*, medium-scale linear programming and large-scale linear programming. Iris chooses the large-scale option which uses an interior-point method and requires *A* and *Aeq* be sparse.

Sparse matrices are especially suitable for expressing the layout constraints, because all constraints are either in the form $x_0 - x_1 >= b$ or in the form $x_0 - x_1 = b$ so that each row of *A* and *Aeq* contains one 1 and one −1, and all other entries are 0. The function *spalloc* is used to

allocate space for sparse matrix. *A = spalloc(m, n, nzmax)* creates an all zero sparse matrix *A* of size *m* x *n* with space to hold *nzmax* nonzeros. The matrix can then be generated entry by entry without requiring repeated storage allocation as the number of nonzeros grows

The objective function is defined as,

$$f = \sum_{rectangles} weight * [(right - left) + (top - bottom)]$$

weight can be different based on different choices. Different layers can have different weights, for example, diffusion can be given a higher weight than contact to get a higher priority. Different nets can have different weights, so that the user can choose to optimize the nets having longer wires with higher priority, or to optimize the nets having more pins with higher priority, or to optimize the nets that the user thinks are more critical with higher priority. Minimizing all the rectangles is equivalent to minimizing total wire length. Besides the four variables related to the rectangles, other variables can also be added to the objective function, for example, the variables representing the boundaries of each layers, so that the total perimeter can be minimized.

Iris creates an m-file named *circuit.m*, which is passed to and executed in Matlab. Matlab runs as a separate process spawned by the Java runtime environment. The solution file named *circuit.dat* is produced by Matlab, and is read by Java object. Below is a typical *circuit.m* file.

```
% size of A matrix
m = 550;
n = 649;
A = spalloc(550,649,1100);
A(1,2) = 1;
A(1,1) = -1;
... ...
```

```
% A*X >= B
B = [ ...
    9 ...
    0 ...
    ... ...
  ];
B = B';
% min C'*X
C = [ ...
    -0.3132465325636185 ...
    0.0 ...
    ... ...
  ];
C = C';
% solve lp
[soln, cost, exitFlag, output] = linprog(C, -A, -B, [], [], zeros(n,1), Inf*ones(n,1));
% write solution to disk
fid = fopen('circuit.dat', 'w');
fprintf(fid, 'cost = %f\n', cost);
fprintf(fid, 'exitFlag = %f\n', exitFlag);
fprintf(fid, 'output = %f %f %s \n', output.iterations, output.cgiterations, output.algorithm);
fprintf(fid, '%f\n', soln);
fclose(fid);
quit;
```

3.3.2 Constraint Graph Based Algorithm Solver

The Matlab linear programming solver either gives an optimal solution when *linprog* converges, or shows the problem is infeasible. The latter indicates an error in the router, and more information is desirable to assist debugging. When there are cycles of constraints leading to infeasibility, the developer wants to know which constraints are in the cycle. When the optimal

solutions are obtained, the developer wants to check the critical path in the final layout to see if it can be improved further. Furthermore, Matlab is slow especially for large designs. Alternative approaches that can give more information to the developer for analysis are desirable.

The layout problem contains only difference constraints, that is, each row of matrix A contains one 1 and one −1, and all other entries are 0. The system of difference constraints can be interpreted as a constraint graph. Given a system $Ax \leq b$ of difference constraints, where A is an m x n matrix, the corresponding constraint graph is a weighted, directed graph $G=(V, E)$, where

$$V = \{ v_0, v_1, \ldots, v_{n+1} \}$$

and

$$E = \{ (v_i, v_j) : x_j - x_i <= b_k \text{ is a constraint, } i, j = 1 \ldots n \}$$
$$\cup \{ (v_0, v_1), (v_0, v_2), \ldots, (v_0, v_n) \} \quad // \text{ create a leftmost vertex, } v_0$$
$$\cup \{ (v_1, v_{n+1}), (v_2, v_{n+1}), \ldots, (v_n, v_{n+1}) \} \quad // \text{ create a rightmost vertex, } v_{n+1}$$

Each vertex v_i, for $i = 1 \ldots n$, corresponds to one unknown variable. Each directed edge corresponds to one inequality. Two additional vertices v_0 and v_{n+1} are added to the constraint graph. Every vertex v_i, for $i = 1 \ldots n$, can be reached from v_0 and v_{n+1} can be reached from every vertex v_i, for $i = 1 \ldots n$. The weight of edge (v_i, v_j) is $w(v_i, v_j) = b_k$. The weight of edges leaving v_0 and edges going into v_{n+1} are 0.

The critical path is the longest path in the constraint graph from v_0 to v_{n+1}. The weight of the longest path is defined as,

$$weight(critical\ path) = \max \{ weight(path) : \text{the path is from } v_0 \text{ to } v_{n+1} \}.$$

Finding the longest path can be turned into the problem of finding the shortest path by changing the constraints in the following way,

$$v_i - v_j \geq b_k (b_k \geq 0) \rightarrow v_j - v_i \leq -1 * b_k (b_k \geq 0)$$

The Bellman-Ford algorithm can solve the single-source-single-destination shortest path problem allowing the edges to have negative weights. The algorithm returns a boolean value indicating whether or not there is a negative weight cycle that is reachable from the single source. If there is a negative weight cycle, then no feasible solution exists. If there is no such cycle, the algorithm produces the shortest path and its weight, and gives a feasible solution.

I use the notation and description of the algorithm in [CLR 90]. The Bellman-Ford algorithm runs in time $O(VE)$ with V vertices and E edges. For the system of difference constraints, this time is $O(n^2+nm)$ with n unknowns and m constraints. With a little modification to the subprogram Initialize-Single-Source (G, s), the algorithm can run in $O(nm)$ time.

Modified-Initialize-Single-Source (G, s)
    For each vertex $v \in V(G)$
        Do $d[v] = 0$, // the original algorithm is $d[v] = \infty$
            $predecessor[v]$ = null
    $d[s] = 0$
End

When the algorithm returns *true*, $d[v]$ gives the feasible solution for vertex v, and the chain of predecessors originating at the single-destination vertex v_{n+1} and running backwards gives the critical path. The algorithm can be modified to find the negative weight cycle when it returns *false*.

Modified-Bellman-Ford (G, w, s)
    Initialize-Single-Source (G, s)
    For i = 1 to $|V(G)| - 1$
        Do for each edge $(u,v) \in E(G)$
            Do Relax(u, v, w) // Relax(u,v,w) is same as described
                                     // in [CLR 90]

feasible = true

For each edge $(u, v) \in E(G)$ and $u<>s$

Do if $d[v] > d[u] + w(u, v)$

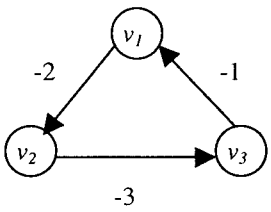
$d[v] = 1$

feasible = false

Return feasible

End

Those vertices having $d[v] = 1$ form the negative weight cycle. The negative weight cycle in the constraint graph indicates that there is a cycle of infeasible constraints in the layout constraint set. The following example shows this situation.



$v_2 - v_1 <= -2$
$v_3 - v_2 <= -3$
$v_1 - v_3 <= -1$

negative weight cycle in constraint graph

$v_1 - v_2 >= 2$
$v_2 - v_3 >= 3$
$v_3 - v_1 >= 1$

cycle of constraints in constraint set

Figure 3.32 Negative Weight Cycle in Constraint Graph

3.3.3 Aspect Oriented Programming

For any user design (i.e., program), Iris should generate a feasible set of constraints. However, in the course of developing Iris, programming errors often led to infeasible systems of constraints. To aid the debugging, it was helpful to be able to identify these infeasible cycle automatically. The next step is to detect where in the source code these constraints are added to the constraint set. Figure 3.33 shows the UML for those classes and their methods where constraints are added.

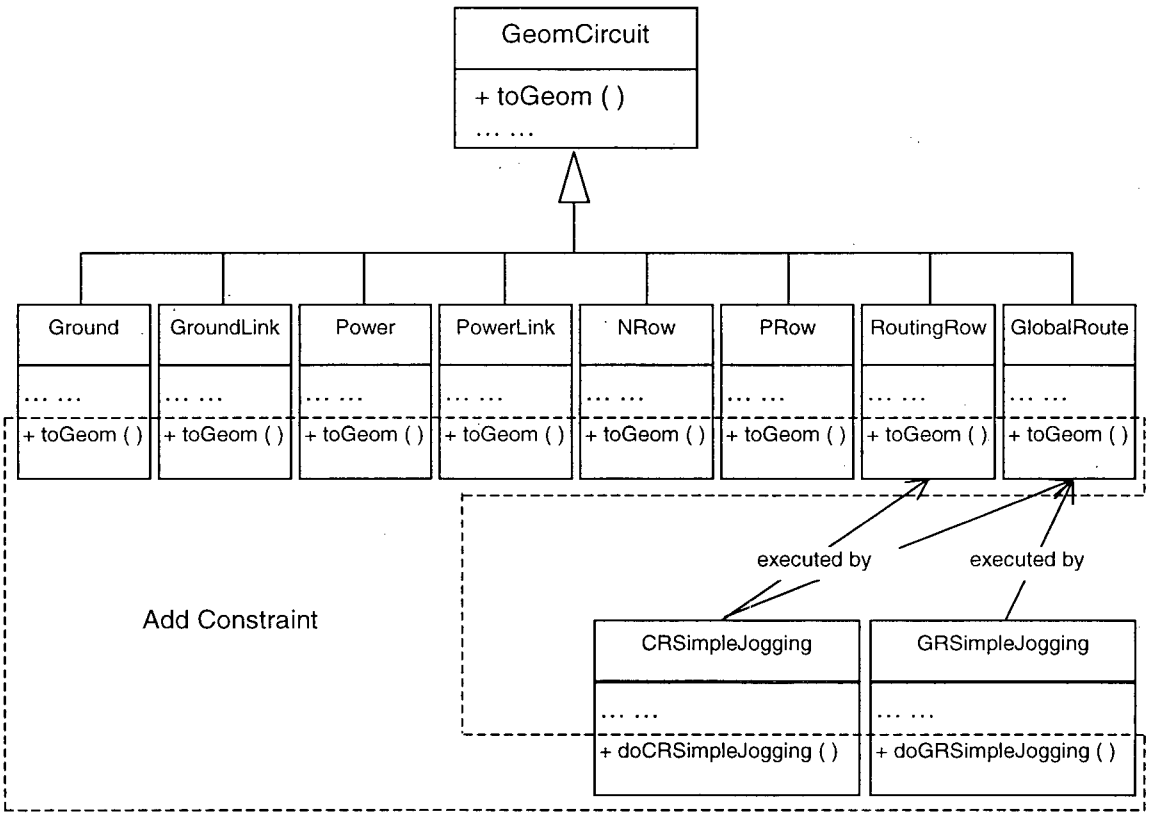


Figure 3.33 UML for Classes Related to Adding Constraint

The method *addConstraint* in class *GeomRowProto* is used to add a constraint to the constraint set. Several classes call *addConstraint* method many times in their method *toGeom* shown in figure 3.33. One way to trace the source code is to print something whenever this method is called. For example, adding one line of code *System.out.println("addConstraint is called at " + classname + linenumber)* before *addConstraint* whenever *addConstraint* is called. The programmer will have to write many such kind of code in several classes. Even if the programmer uses a subprogram to print debugging message, the *classname* and *linenumber* has to be passed to the subprogram whenever *addConstraint* is called. Either way the programmer has to explicitly do something to trace the code. Thus the code for adding constraint and the code

for tracing the cycle of constraints tangles together which makes the code a mess. Aspect oriented programming helps in this situation.

Aspect oriented programming [Elrad 01] provides mechanisms -- aspects, to localize crosscutting concerns and implement the system in a crosscutting way. Tracing the constraints is one concern, and adding constraints is another concern. Separating these two concerns reflects the way the developer wants to think about the system. This separation localizes different concerns, hence makes the design and code more modular.

Aspect oriented programming builds on existing technologies. I used AspectJ [Kiczales 01], an aspect oriented extension to Java, in Iris. To trace such a constraint $v_1 - v_0 >= 1$, the tracing aspect will be:

```
aspect Tracing {
        pointcut trace_addConstraint(Variable v1, Variable v2, double d):
        (call(void GeomRowProto.addConstraint(Variable, Variable, double)) && args(v1, v2, d));

        before(Variable v1, Variable v2, double d): trace_addConstraint(v1, v2, d) {
                if (v1.id() == 0 && v2.id() == 1 && d == 1) {
                // v1.id() == 0 represents v0 and v2.id() == 1 represents v1
                        SourceLocation l = thisJoinPoint.getSourceLocation();
                        System.out.println(l.getFileName() + " " + l.getLine() +
                        " [within " + l.getWithinType() + "] ");
                }
        }
}
```

In this aspect, the pointcut *trace_addConstraint* identifies any call to *addConstraint* method defined by *GeomRowProto*. Before advice *before(...): trace_addConstraint(...)* prints a message showing at which line of which class file the method *addConstraint* is called when the join point is reached. For example, this message could be "PRow.java 176 [within class Circuit.PRow]". It tells that at line 176 of the file *PRow.java* within the class *Circuit.PRow* the

method *addConstraint* is called, that is, the constraint is added to the constraint set. Then the developer can check the source code at line 176 of file *PRow.java* to see whether or not the constraint should be added.

With the Java language, the developers have to delete or comment out a lot of tracing code when debugging is complete if they want to disable the tracing, which is frustrating. AspectJ provides an easy way to remove the debugging message by removing the aspects from the compile configuration when they are not needed, that is, changing *makefile* not to include the tracing aspects. Since AspectJ is used only for debugging in Iris, the user doesn't need to know anything about AspectJ. Once the *makefile* is changed, Iris can be built and run in the traditional Java environment without the support of AspectJ.

Chapter 4

Experiments

Several examples were tested to check several design goals. The testing was run on a Linux machine with 1GHz CPU, 256M RAM, ajc version 1.0.6 running on java 1.4.0_01. The objective function was designed to minimize layout perimeter and total wire length. The weights assigned to the layers are listed in table 4.1.

layer	pdiff	ndiff	ptran	ntran	pdc	ndc	poly	pc	metal1	metal2	m2c
weight	10	10	1	1	1	1	3	4	1	1	1

Table 4.1 The Weight of Layers

The first design goal is to make the Iris generated layout comparable to the hand layout. The comparison was performed between an Iris generated layout and a manually designed layout under technology MOSIS SCMOS with the example FifoStage. The manual layout is shown in figure 4.1. The Iris generated layout is shown in figure 4.1, which is design rule correct for MOSIS SCMOS technology. The manual layout included the input and output signals, p-well and n-well which are not yet implemented in Iris. This leads to the Iris generated layout that is

smaller than the manual layout. Considering the extra area required by input and output signals, p-well and n-well which are not yet included in the Iris generated layout, the result is still encouraging, which demonstrates that Iris generated layout is comparable to the manual layout.

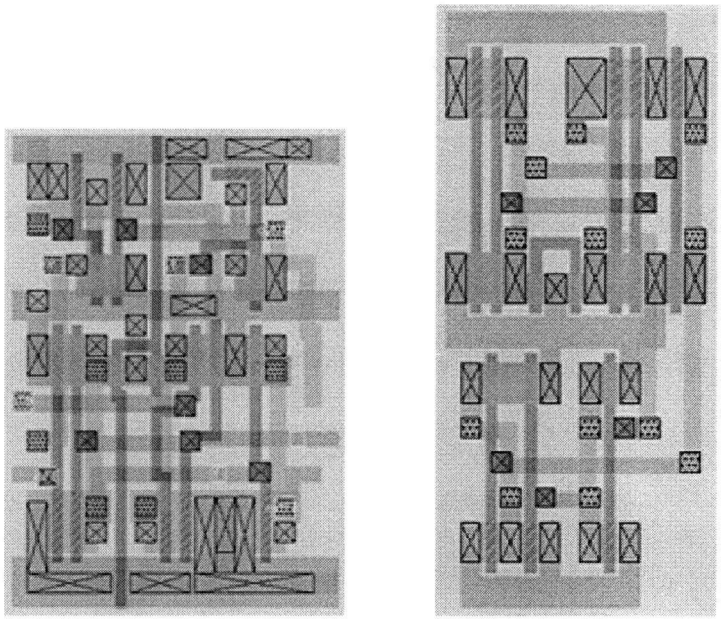


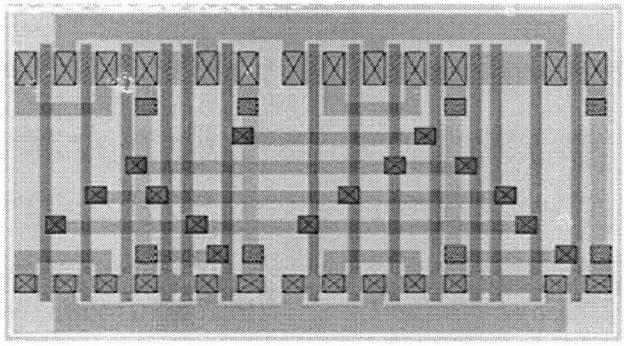
Figure 4.1 FifoStage Manual Layout (left) and Iris Generated Layout (right)

The second goal is to achieve designer efficiency. From the n-bit ripple-carry adder example, we can see the user may need to spend some time in writing the netlist for the leaf cell. Other parts of the code don't need much time because they are similar for different cells, so the user may reuse other parts of code for different cells. Designer efficiency is greatly improved over manual layout especially for upper level cells such as the one-bit full adder and the n-bit ripple-carry adder which require only a few lines of code in Iris but require a lot of time by hand layout. Further studies with real users are needed to quantify theses claims.

Most of the execution time is spent on constraint solving, and for smaller circuits the time is mostly spent starting Matlab. The number of transistors determines the size of the

constraint matrix, which determines the execution time of Matlab's linear program solver. The number of transistors and the size of constraint matrix are in a roughly linear relationship. With the increasing of the number of transistors, the execution time increases significantly. Table 4.2 gives the excution time of all the examples. Figure 4.5 shows the relationship of the number of transistors and the execution time. I drew this figure using a polynomial of degree n = 2 to fit the data. The result polynomial is $p(x) = 0.0368x^2 + 0.6735x + 10.0327$.

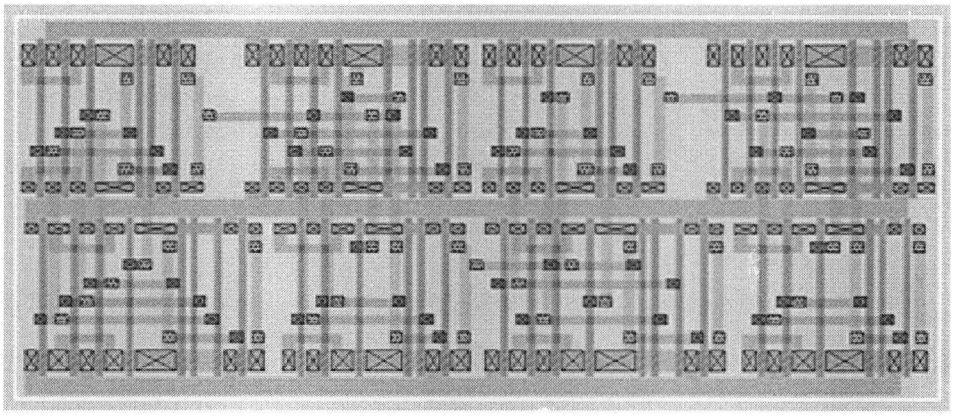
The third goal is to achieve flexibilty. The flexibility is implemented through the placement methods that the circuit may use. In the leaf cell, the transistor can be put using the *flip* method to implement drain/source sharing and diffusion sharing. In the upper level cell, the user can use the *left* and *above* methods to build a module from one or more leaf cells arbitrarily. All the cells can be flipped either horizontally or vertically by using the *mirrorX* or *mirrorY* method respectively. In chapter 3, we described how to build the carry block and the sum block; how to build a one-bit full adder from the carry block and the sum block using the *above* method; and how to build an n-bit ripple-carry adder from n copies of the one-bit full adder using the *left* method. The following example shows the flexibility in which the one-bit full adder was built from the carry block and the sum block using the *left* method. Only one line of code in class OneBitFA in figure 3.14 needs to be changed. The result is shown in figure 4.2.



c= cb..circuit().above(sb.circuit()) ⟹ cb.circuit().left(sb.circuit())

Figure 4.2 One-Bit Full Adder

Here is another example to show the flexibility. Observing the n-bit ripple-carry adder shown in figure 3.19, we can see that it is not a balanced layout in terms of space utilization. The upper part of the layout has a lot of white space while the lower part of the layout is quite dense. This is because the sum block is wider than the carry block. Putting 4 copies of the carry block side by side in the upper part and 4 copies of the sum block side by side in the lower part, this difference of area utilization is increased. Trying to get a more balanced layout, one good way would be vertically flipping every other one-bit full adder. The code change to the n-bit ripple-carry adder in figure 3.18 and the layout are shown below. The ratio of the layout before and after the flip is 1:09:1.



```
c = c.left(fa.circuit());
```

```
if (i % 2 != 0)
    c = c.left(fa.circuit().mirrorY());
else
    c = c.left(fa.circuit());
```

Figure 4.3 n-Bit Ripple Carry Adder

The fourth goal is to support multiple technologies. Currently Iris supports MOSIS SCMOS and TSMC 0.18-micron design technology. All the above examples are based on MOSIS SCMOS technology. The following figure shows FifoStage under TSMC 0.18-micron technology. A lot of optimization still needs to be done under this technology.

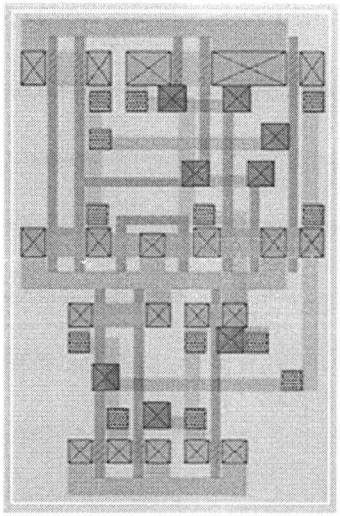


Figure 4.4 FifoStage under TSMC 0.18-micron Technology

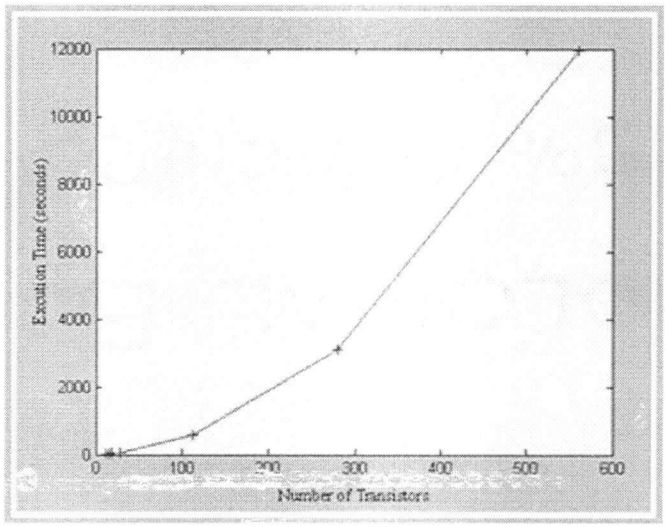


Figure 4.5 The Relationship of Number of Transistors and Execution Time

Example	Height (λ)	Width (λ)	Cell Area (λ^2)	Number of Transistors	Execution Time (seconds)	Constraint Matrix Size
Manual FifoStage (Figure 4.1)	94	66	6204	18	X	X
FifoStage under MOSIS SCMOS Technology (Figure 4.1)	119	52	6188	18	47	550*649
FifoStage under TSMC 0.18-micron Technology (Figure 4.4)	165	102	16830	18	46	539*654
Carry (Figure 3.6)	61	49	2989	12	30	318*375
Sum (Figure 3.12)	68	65	4420	16	34	372*394
One-Bit Full Adder (Figure 3.15)	137	80	10960	28	64	798*703
n-Bit Ripple-Carry Adder (n=4) (Figure 3.19)	138	337	46506	112	492	2845*1186
n-Bit Ripple-Carry Adder (n=4) (Figure 4.3)	144	297	42768	112	566	2916*1193
n-Bit Ripple-Carry Adder (n=10)	144	750	108000	280	3118	7233*2168
n-Bit Ripple-Carry Adder (n=20)	145	1505	218225	560	11932	14428*3793

Table 4.2 Experimental Results

Chapter 5

Conclusions and Future Work

This thesis describes the automatic circuit layout generation tool Iris. Iris views the circuit from the point of view of object oriented programming. Classes and methods written in Java are designed in such a way that the circuit level abstraction is reflected. Aspect oriented programming techniques are applied in the development of Iris. Simple heuristic methods are proposed to solve the routing problem. Constraint solving methods used in this thesis produce the optimal solution. New design technologies are easy to integrated into Iris. The layouts generated by Iris are comparable in area to full custom layouts with great savings in designer effort and time. Layout modifications are greatly simplified for the designer.

Different approaches have been proposed to solve the layout problems. Much current research addresses layout design automation. As for Iris per se, the future work will be in the following areas.

By checking the layout generated by Iris, we can see that some layouts can be compacted further. Currently Iris uses some simple methods for routing. In the future, we may try more sophisticated heuristics and algorithms.

The layout design process involves many steps. The quality of results obtained in a later step depends on the quality of results obtained in the earlier steps. Hence the earlier steps have more influence on the quality of the final layout. The first step of Iris is the placement, which is based on the relative positions defined in the user interface file. Optimizing the relative position is necessary for generating a good layout.

Fabrication technologies are changing toward supporting more process features, such as more metal layers. Currently Iris supports routing on metal1 and metal2 layers. Supporting more metal layers is desirable for producing more compact designs. With more metal layers available for routing, three dimensional routing techniques are necessary to achieve the performance and density goals. Over-the-cell routing is one of such techniques for reducing the routing area.

Future work can also be done in the area of supprting more architectural styles, such as transistor folding, transistor stacking and via stacking.

Bibliography

[CLR 90] T. H. Cormen, C. E. Leiserson and R. L. Rivest. "Introduction to Algorithms". 1990.

[Deutsch 76] D. N. Deutsch. "A Dogleg Channel Router". *Proceedings of 13th ACM/IEEE Design Automation Conference*, 1976.

[Elrad 01] T. Elrad, R. E. Filman, A. Bader. "Aspect-Oriented Programming". *Communications of the ACM*, Vol. 44, No. 10, October, 2001.

[Garey 77] M. R. Garey and D. S. Johnson. "The Rectilinear Steiner Tree Problem is NP-Complete". *SIAM Journal Applied Mathematics*, 1977

[Hashimoto 71] A. Hashimoto and J. Stevens. "Wire Routing by Optimization Channel Assignment within Large Apertures". *Proceedings of the 8th Design Automation Workshop*, 1971.

[Hsu 83] C. P. Hsu. "General River Routing Algorithm". *Proceedings of 20th Design Automation Conference*, June, 1983.

[Hwang 76] F. K. Hwang. "On Steiner Minimal Trees with Rectilinear Distance". *SIAM Journal of Applied Mathematics*, January, 1976.

[Kiczales 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kerstan, J. Palm, W. G. Griswold. "Getting Started with AspectJ". *Communications of the ACM*, Vol. 44, No. 10, October, 2001.

[Kruskal 56] J. B. Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem". *Proceddings of the American Mathematical Society*, 1956.

[Lin 01] Lan Lin. "An Auomatic Layout Generator for Integrated Circuit Design". MSc thesis. 2001

[Ousterhout 84] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G. Taylor. "Magic: A VLSI Layout System". *Proceedings of the 21st Design Automation Conference*, 1984.

[Park 00] Soondal Park. "LPABO ver 5.72 User Manual (2000.5.2)". http://orlab.snu.ac.kr/pub/software/lpabo/lpabo572.wp

[Prim 57] R. C. Prim. "Shortest Connection Networks and Some Generalizations". *Bell System technical Journal*, 1957.

[Rivest 82] R. L. Rivest and C. M. Fiduccia. "A Greedy Channel Router". *Proceedings of the 19th Design Automation Conference*, 1982.

[Szymanski 85] T. G. Szymanski. "Dogleg Channel Routing is NP-Complete". *IEEE Transactions on Computer-Aided Design*, January, 1985.

Appendix A

User Interface Example Source

Table A.1 Index of User Interface Example Source

```
import Circuit.*;
import java.util.*;
import Constraints.*;

public class CarryBlock {
   private DefaultSignal _Vdd;
   private Signal A;  public Signal A() { return A; }
   private Signal B;  public Signal B() { return B; }
   private Signal Cin;  public Signal Cin() { return Cin; }
   private Signal Not_Cout;  public Signal Not_Cout() { return Not_Cout; }
   private Signal Cout;  public Signal Cout() { return Cout; }
   private Circuit c;  public Circuit circuit() { return c; }
   private Power Vdd;  public Signal Vdd() { return(Vdd); }
   private Ground Gnd;  public Signal Gnd() { return Gnd; }
   private PRow pr;
   private NRow nr;

   public CarryBlock() {
         _Vdd = new DefaultSignal();
         A = new DefaultSignal();
         B = new DefaultSignal();
         Cin = new DefaultSignal();
         Not_Cout = new DefaultSignal();
         Cout = new DefaultSignal();
         Vdd = new Power(_Vdd, 6);
         Gnd = new Ground(6);

         // Based on P-Transistor Row Layout Model
         boolean[] pflip = new boolean[] { true, false, false, true, true, false };
         PTransistor[] p = new PTransistor[pflip.length];
         for (int i = 0; i < p.length; i++) {
            p[i] = new PTransistor(8);
            if (pflip[i]) p[i] = p[i].flip();
         }

         // Based on N-Transistor Row Layout Model
         boolean[] nflip = new boolean[] { false, true, true, false, false, true };
         NTransistor[] n = new NTransistor[nflip.length];
         for (int i = 0; i < n.length; i++) {
            n[i] = new NTransistor(4);
            if (nflip[i]) n[i] = n[i].flip();
         }

         // Netlist
         p[0].source().connect(p[1].source()).connect(p[4].source())
            .connect(p[5].source()).connect(Vdd);
```

```
        n[0].drain().connect(n[1].drain()).connect(n[4].drain())
          .connect(n[5].drain()).connect(Gnd);
        p[0].gate().connect(p[4].gate()).connect(n[0].gate())
          .connect(n[4].gate()).connect(A);
        p[1].gate().connect(p[3].gate()).connect(n[1].gate())
          .connect(n[3].gate()).connect(B);
        p[2].gate().connect(n[2].gate()).connect(Cin);
        p[0].drain().connect(p[1].drain()).connect(p[2].source());
        n[0].source().connect(n[1].source()).connect(n[2].drain());
        p[2].drain().connect(n[2].source()).connect(p[3].drain())
          .connect(n[3].source()).connect(p[5].gate())
          .connect(n[5].gate()).connect(Not_Cout);
        p[3].source().connect(p[4].drain());
        n[3].drain().connect(n[4].source());
        p[5].drain().connect(n[5].source()).connect(Cout);

        // Based on P-Transistor Row Layout Model
        for (int i = 0; i < p.length; i ++) {
          if (i == 0) pr = p[i];
          else pr = pr.left(p[i]);
        }

        // Based on N-Transistor Row Layout Model
        for (int i = 0; i < n.length; i ++) {
          if (i == 0) nr = n[i];
          else nr = nr.left(n[i]);
        }

        // Based on Carry Block Circuit Layout Model
        c = (Vdd)
          .above(pr)
          .above(nr)
          .above(Gnd);
    }
}
```

```
import Circuit.*;
import java.util.*;
import Constraints.*;

public class SumBlock {
   private DefaultSignal _Vdd;
   private Signal A;  public Signal A() { return A; }
   private Signal B;  public Signal B() { return B; }
   private Signal Cin;  public Signal Cin() { return Cin; }
   private Signal Sum;  public Signal Sum() { return Sum; }
   private Signal Not_Cout;  public Signal Not_Cout() { return Not_Cout; }
   private Circuit c;  public Circuit circuit() { return c; }
   private Power Vdd;  public Signal Vdd() { return(Vdd); }
   private Ground Gnd;  public Signal Gnd() { return Gnd; }
   private PRow pr;
   private NRow nr;

   public SumBlock() {
         _Vdd = new DefaultSignal();
         A = new DefaultSignal();
         B = new DefaultSignal();
         Cin = new DefaultSignal();
         Sum = new DefaultSignal();
         Not_Cout = new DefaultSignal();
         Vdd = new Power(_Vdd, 6);
         Gnd = new Ground(6);

         // Based on P-Transistor Row Layout Model
         boolean[] pflip = new boolean[] { false, true, false, false, true, true, true, false };
         PTransistor[] p = new PTransistor[pflip.length];
         for (int i = 0; i < p.length; i++) {
            p[i] = new PTransistor(8);
            if (pflip[i]) p[i] = p[i].flip();
         }

         // Based on N-Transistor Row Layout Model
         boolean[] nflip = new boolean[] { true, false, true, true, false, false, false, true };
         NTransistor[] n = new NTransistor[nflip.length];
         for (int i = 0; i < n.length; i++) {
            n[i] = new NTransistor(4);
            if (nflip[i]) n[i] = n[i].flip();
         }

         // Netlist
         p[0].source().connect(p[1].source()).connect(p[2].source())
            .connect(p[6].source()).connect(p[7].source()).connect(Vdd);
```

```
        n[0].drain().connect(n[1].drain()).connect(n[2].drain())
           .connect(n[6].drain()).connect(n[7].drain()).connect(Gnd);
        p[0].gate().connect(p[6].gate()).connect(n[0].gate())
           .connect(n[6].gate()).connect(A);
        p[1].gate().connect(p[5].gate()).connect(n[1].gate())
           .connect(n[5].gate()).connect(B);
        p[2].gate().connect(p[4].gate()).connect(n[2].gate())
           .connect(n[4].gate()).connect(Cin);
        p[0].drain().connect(p[1].drain()).connect(p[2].drain())
           .connect(p[3].source());
        n[0].source().connect(n[1].source()).connect(n[2].source())
           .connect(n[3].drain());
        p[3].gate().connect(n[3].gate()).connect(Not_Cout);
        p[4].source().connect(p[5].drain());
        p[5].source().connect(p[6].drain());
        n[4].drain().connect(n[5].source());
        n[5].drain().connect(n[6].source());
        p[3].drain().connect(n[3].source()).connect(p[4].drain())
           .connect(n[4].source()).connect(p[7].gate()).connect(n[7].gate());
        p[7].drain().connect(n[7].source()).connect(Sum);

        // Based on P-Transistor Row Layout Model
        for (int i = 0; i < p.length; i ++) {
           if (i == 0) pr = p[i];
           else pr = pr.left(p[i]);
        }

        // Based on N-Transistor Row Layout Model
        for (int i = 0; i < n.length; i ++) {
           if (i == 0) nr = n[i];
           else nr = nr.left(n[i]);
        }

        // Based on Sum Block Circuit Layout Model
        c =  (Vdd)
           .above(pr)
           .above(nr)
           .above(Gnd);
    }
}
```

```
import Circuit.*;
import java.util.*;
import Constraints.*;

public class OneBitFA {
   private DefaultSignal _Vdd;
   private Signal A;  public Signal A() { return A; }
   private Signal B;  public Signal B() { return B; }
   private Signal Cin;  public Signal Cin() { return Cin; }
   private Signal Sum;  public Signal Sum() { return Sum; }
   private Signal Not_Cout;  public Signal Not_Cout() { return Not_Cout; }
   private Signal Cout;  public Signal Cout() { return Cout; }
   private Circuit c;  public Circuit circuit() { return c; }
   private Power Vdd;  public Signal Vdd() { return(Vdd); }
   private Ground Gnd;  public Signal Gnd() { return Gnd; }

   public OneBitFA() {
         _Vdd = new DefaultSignal();
         A = new DefaultSignal();
         B = new DefaultSignal();
         Cin = new DefaultSignal();
         Sum = new DefaultSignal();
         Not_Cout = new DefaultSignal();
         Cout = new DefaultSignal();
         Vdd = new Power(_Vdd, 6);
         Gnd = new Ground(6);

         // build the Carry Block and the Sum Block individually
         SumBlock sb = new SumBlock();
         CarryBlock cb = new CarryBlock();

         // make signal conections between the two blocks
         sb.Vdd().connect(cb.Vdd()).connect(Vdd);
         sb.Gnd().connect(cb.Gnd()).connect(Gnd);
         A = sb.A().connect(cb.A());
         B = sb.B().connect(cb.B());
         Cin = sb.Cin().connect(cb.Cin());
         Not_Cout = sb.Not_Cout().connect(cb.Not_Cout());

         // the output signals of the one-bit full adder
         Cout = cb.Cout();
         Sum = sb.Sum();
         c = cb.circuit().above(sb.circuit());
   }
}
```

```
import Circuit.*;
import java.util.*;
import Constraints.*;

public class nBitRCAdder {
   private DefaultSignal _Vdd;  public Signal Vdd() { return(_Vdd); }
   private Signal[] A;  public Signal[] A() { return A; }
   private Signal[] B;  public Signal[] B() { return B; }
   private Signal Cin;  public Signal Cin() { return Cin; }
   private Signal[] Sum;  public Signal[] Sum() { return Sum; }
   private Signal Cout;  public Signal Cout() { return Cout; }
   private Circuit c;  public Circuit circuit() { return c; }
   private Power Vdd;
   private Ground Gnd;  public Signal Gnd() { return Gnd; }

   public nBitRCAdder(int n) {
         _Vdd = new DefaultSignal();
         Vdd = new Power(_Vdd, 6);
         Gnd = new Ground(6);
         A = new Signal[n];
         B = new Signal[n];
         Sum = new Signal[n];
         OneBitFA fa = new OneBitFA();
         c = fa.circuit();
         A[0] = fa.A();
         B[0] = fa.B();
         Sum[0] = fa.Sum();
         Cin = fa.Cin();
         Cout = fa.Cout();
         fa.Vdd().connect(Vdd);
         fa.Gnd().connect(Gnd);

         // build one-bit full adder repeatedly
         for (int i = 1; i < n; i ++) {
            fa = new OneBitFA();
            fa.Cin().connect(Cout);
            fa.Vdd().connect(Vdd);
            fa.Gnd().connect(Gnd);
            c = c.left(fa.circuit());
            A[i] = fa.A();
            B[i] = fa.B();
            Sum[i] = fa.Sum();
            Cout = fa.Cout();
         }
   }
```

```
    public static void main(String[] args) throws Exception {
        Technology.setTech(Scmos.tech());
        int solver = GeomCircuit.Matlab;
        ObjFunc[] obj = new ObjFunc[1];
        obj[0] = new ObjFunc();
        obj[0].setWeight(ObjFunc.boundsWt, 0);
        obj[0].setWeight(ObjFunc.layerWt, 5);
        obj[0].setWeight(ObjFunc.netWt, 0);

        (new GeomCircuit((new nBitRCAdder(4)).circuit(), Technology.tech(), "nBitRCAdder",
solver, obj)).toGeomCircuit();
    }
}
```

```
import Circuit.*;
import java.util.*;
import Constraints.*;

public class FifoStage {
   public static void main(String[] args) {

   long begin = System.currentTimeMillis();
   System.out.println("time : " + begin);

   Technology.setTech(Scmos.tech());
   ObjFunc[] obj = new ObjFunc[1];
   obj[0] = new ObjFunc();
   int solver = GeomCircuit.Matlab;

   DefaultSignal Vdd = new DefaultSignal();
   Signal inTB = new DefaultSignal();
   Signal inFB = new DefaultSignal();
   Signal outTB = new DefaultSignal();
   Signal outFB = new DefaultSignal();
   Power vdd1 = new Power(Vdd, 6);
   Power vdd2 = new Power(Vdd, 6);
   Ground gnd = new Ground(6);

   PTransistor[] t0 = {new PTransistor(8),new PTransistor(8),new PTransistor(8)};
   NTransistor[] t1 = {new NTransistor(8),new NTransistor(8),new NTransistor(8)};
   NTransistor[] t2 = {new NTransistor(10),new NTransistor(10),new NTransistor(6),
         new NTransistor(6),new NTransistor(10),new NTransistor(10),new NTransistor(10)};
   PTransistor[] t3 = {new PTransistor(12),new PTransistor(12),new PTransistor(12),
         new PTransistor(12),new PTransistor(12)};

   t0[0].source().connect(t0[1].drain()).connect(t0[2].drain()).connect(vdd1);
   t0[0].drain().connect(t0[1].source()).connect(t1[0].source());
   t0[0].gate().connect(t1[0].gate()).connect(t2[6].drain()).connect(t3[4].drain()).connect(outFB);
   t0[1].gate().connect(t0[2].source()).connect(t1[1].gate())
      .connect(t1[2].source()).connect(outTB);
   t0[2].gate().connect(t1[2].gate()).connect(t2[3].drain())
      .connect(t2[4].source()).connect(t3[2].source());
   t1[0].drain().connect(t1[1].source());
   t1[1].drain().connect(t1[2].drain()).connect(t2[0].source())
      .connect(t2[2].drain()).connect(t2[3].source())
      .connect(t2[5].drain()).connect(t2[6].source()).connect(gnd);
   t2[0].gate().connect(t3[0].gate()).connect(inFB);
   t2[0].drain().connect(t2[1].source());
   t2[1].gate().connect(t2[5].gate()).connect(t3[1].gate()).connect(t3[3].gate());
   t2[1].drain().connect(t2[2].source()).connect(t2[6].gate())
```

```
    .connect(t3[1].drain()).connect(t3[4].gate());
  t2[2].gate().connect(t2[3].gate());
  t2[4].gate().connect(t3[2].gate()).connect(inTB);
  t2[4].drain().connect(t2[5].source());
  t3[0].source().connect(t3[3].drain()).connect(t3[4].source()).connect(vdd2);
  t3[0].drain().connect(t3[1].source());
  t3[2].drain().connect(t3[3].source());

  PRow pRow1 = t0[0].left(t0[1]).left(t0[2]);
  PRow pRow2 = t3[0].left(t3[1]).left(t3[2]).left(t3[3]).left(t3[4]);
  NRow nRow1 = t1[0].left(t1[1]).left(t1[2]);
  NRow nRow2 = t2[0].left(t2[1]).left(t2[2]).left(t2[3]).left(t2[4]).left(t2[5]).left(t2[6]);

  Circuit fifoStage =  (vdd2)
    .above(pRow2)
    .above(nRow2)
    .above(gnd)
    .above(nRow1)
    .above(pRow1)
    .above(vdd1);

  obj[0].setWeight(ObjFunc.boundsWt, 0);
  obj[0].setWeight(ObjFunc.layerWt, 5);
  obj[0].setWeight(ObjFunc.netWt, 0);

  GeomCircuit geomCircuit = new GeomCircuit(fifoStage, Technology.tech(), "FifoStage", solver,
obj);
  geomCircuit.toGeomCircuit();
  }
}
```