

Designing an Automatic Schematic Generator for a Netlist Description

C.R. Lageweg

Laboratory of Computer Architecture and Digital
Techniques (CARDIT)
Department of Electrical Engineering
Faculty of Information Technology and Systems

Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft,
The Netherlands

September 8, 1998

Delft University of Technology
Faculty of Information Technology and Systems

Type : Technical Report
Number of pages : 83
Date : September 8, 1998

Lab./Dept. : Laboratory of Computer Architecture and Digital Techniques (CARDIT)
Codenummer : 1-68340-44(1998)03
Author : C.R. Lageweg
Title : **Designing an Automatic Schematic Generator for a Netlist Description**

Supervisor : Prof. Dr. Ir. A.J. van de Goor
Mentor : Dr. Ir. J.Th. van der Linden
Ir. M.H. Konijnenburg

Designing an Automatic Schematic Generator for a Netlist Description

Abstract

Within the Computer Architecture and Digital Technique (CARDIT) laboratory of the department of Electrical Engineering at the faculty of Information Technology and Systems research is done in the field of test algorithm development. For this purpose the Delft Advanced Test generation system DAT has been designed; it accepts a netlist description of a digital circuit and generates a test set for this circuit based on different fault models.

Test algorithm development is based on the logic structure of a circuit. To help algorithm designers study parts of the circuit in detail, a transformation from the netlist to a schematic diagram is needed. Netlists are a summation of components and connections. Manually transforming a netlist into a schematic diagram becomes nearly impossible for large circuits. Therefore, an Automatic Schematic Generator (ASG) has been developed.

This report gives a review of existing literature in the field of ASG, an in-depth examination of the existing ASG solution in the DAT framework, and a detailed description of bug fixes, improvements and additions to the original solution.

Contents

ABSTRACT	iii
1 Introduction	1
1.1 The DAT Project	1
1.2 The purpose of an Automatic Schematic Generator	2
1.3 Introduction to the existing Automatic Schematic Generator	2
1.4 Task Assignment Overview	3
1.5 Report Overview	3
2 The Automatic Schematic Generation Problem	5
2.1 General ASG Terminology	5
2.2 The Netlist Description	7
2.3 The Schematic Diagram	8
2.4 General Problem Decomposition	9
2.4.1 Logical Module Placement	10
2.4.2 Logical Connection Routing	11
2.4.3 Geometrical Module Placement	13
2.4.4 Geometrical Connection Routing	13
2.5 General ASG solution Overview	13
3 Survey of existing Literature	15
3.1 "An Algorithm for Automatic Line Routing"	16
3.2 "Efficient Algorithms for Channel Routing"	17

3.3	"HAL: A heuristic Approach to Schematic Generation"	21
3.4	"Placement and Routing for Logic Schematics"	22
3.5	"Computer-Generated Multi-Row Schematics"	23
3.6	"Autodraft: Automatic Synthesis of Circuit Schematics"	24
3.7	"Automatic Generation of Digital System Schematic Diagrams"	26
3.8	"GEMS: An Automatic Layout Tool for MIMOLA Schematics"	27
3.9	"VISION: VHDL Induced Schematic Imaging on Net-Lists"	27
3.10	"From Network to Artwork"	29
3.11	"Structure Optimization in Logic Schematic Generation"	30
3.12	"Schematic Generation with an Expert System"	31
3.13	"ASG: Automatic Schematic Generator"	33
3.14	Conclusions of the Literature Reviews	35
4	The Original ASG Implementation	37
4.1	Starting the ASG	37
4.2	The ASG Input Data	38
4.3	The Graphical Output	39
4.4	ASG Source Code Hierarchy	39
4.5	ASG Specific Data Structures	40
4.6	ASG Functions and Algorithms	42
4.6.1	Logical Placement Algorithms	43
4.6.2	Logical Routing Algorithms	45
4.6.3	Geometrical Placement Algorithms	46
4.6.4	Geometrical Routing Algorithms	47
4.6.5	Fanout Point Algorithms	49
4.7	Conclusions about the Original ASG Implementation	49
5	Improvements of the existing ASG Algorithms	55
5.1	Changes to the Data Structure	55
5.2	Improvements to existing ASG Algorithms	56
5.2.1	Improvements to Logical Placement Algorithms	56
5.2.2	Improvements to Logical Routing Algorithms	59
5.2.3	Improvements to Geometrical Placement Algorithms	60
5.2.4	Improvements to Geometrical Routing Algorithms	60
5.3	Additional ASG Algorithms	63
5.4	Additions to the Graphic Output	66
5.5	Improvements to the Command Line Environment	68
6	Conclusions and Further Research	71
	Bibliography	77
A	ASG related Commands and Settables	79
A.1	ASG related Commands	79
A.2	ASG related Settables	81

List of Figures

2.1	Example of a Passthrough.	6
2.2	Example of a Netlist and its corresponding Schematic.	7
2.3	Example of a Schematic Diagram.	9
2.4	The logical Grid.	10
2.5	Example of a Channel and some Tracks.	12
2.6	Examples of overlap, (a) vertical and (b) horizontal overlap.	12
3.1	The Channel as described in [TY82].	18
3.2	Example of a channel (a) and its corresponding VCG (b) and HCG (c).	19
3.3	The levels in a VCG.	20
3.4	Example of a loop in the VCG graph, (a) a channel and (b) the matching VCG graph.	20
3.5	Example of Net Merging using the VCG, (a) before and (b) after.	21
3.6	Example of the Decrease in Readability when Passthroughs do not have Fixed Logical Positions	26
3.7	The Bubbling Row Assignment Algorithm. (a) Initial values, (b) after value propagation, (c) after row sorting.	34
4.1	Schematic Symbols used in this Manual.	38
4.2	Screen-dump of the ASG's graphic output.	40
4.3	Drawing of the same Schematic with and without fixed Element Area's.	41
4.4	Data structures used for Placement Information.	42
4.5	Data structures used for Routing Information.	43
4.6	The Backward Traversal Algorithm: (a) before and (b) after applying.	44

4.7	The Horizontal Alignment Algorithm, (a) before and (b) after applying.	44
4.8	The Fanin Centering Algorithm, (a) before and (b) after applying.	45
4.9	The Inputs Vertical Adjustment Algorithm, (a) before and (b) after applying. . .	45
4.10	Making room for Passthroughs, (a) before and (b) after applying.	46
4.11	Fine-tuning a Block's Y Coordinate, (a) before and (b) after applying.	47
4.12	An Example of the Track Assignment Algorithm.	48
4.13	Example of Fanout Point Categories.	50
4.14	Example of the Ghost Line problem, (a) the original situation and (b) the correct situation	52
5.1	Row Calculation during Fanin Centering, (a) before and (b) after Improvement.	58
5.2	Slot Assignment Possibilities, (a) before and (b) after Improvement.	58
5.3	Assigning Connections to Tracks, (a) original and (b) corrected Implementation.	60
5.4	Example of multiple Branches from a single Fanout Point Stacked on Top of Each other.	61
5.5	The Stem's horizontal Line Segment.	62
5.6	Examples of the geometrical Routing of Branches.	63
5.7	Example of the Adjusted Bubbles Sort Algorithm.	65
5.8	Example of the pairwise swap algorithm, (a) before and (b) after applying. . . .	65
5.9	The four Cone types. (A) Input cone, (B) Output cone, (C) FFR cone, (D) Region 'cone'.	67
6.1	Circuit 7482 drawn with the original DAT-ASG.	72
6.2	Circuit Inputs in circuit 7482 drawn with the original DAT-ASG	73
6.3	Incorrect Fanin Centering in circuit 7482 drawn with the original DAT-ASG. . .	73
6.4	Horizontal Overlap in circuit 7482 drawn with the original DAT-ASG	74
6.5	Pairwise twisted gates in circuit 7482 drawn with the original DAT-ASG	74
6.6	Circuit 7482 displayed by current DAT-ASG, using Backward Traversal for ini- tial placement.	75
6.7	Circuit 7482 displayed by current DAT-ASG, using Bubble Sort for initial place- ment.	76

Introduction

The research outlined in this technical report is part of the Delft Advanced Test (DAT) project. This chapter will start with a short description of the DAT platform in Section 1.1. The purpose of an Automatic Schematic Generator (ASG) will be explained in Section 1.2. The existing ASG implementation will be briefly introduced in Section 1.3. An overview of the master thesis task assignment will be given in Section 1.4. This chapter ends with an overview of the structure of this report in Section 1.5.

1.1 The DAT Project

With the ongoing decrease of transistor size in the modern chips, and the increasing number of transistors used in a single chip, coupled with the increasing complexity of designs, it has become more and more difficult to ensure the correctness of the final product. The most straightforward way to test a chip is to try every possible input and verify the correctness of the resulting output. Given the time this would take to test a single chip, and the typical large scale production, this has become impossible. This means that other ways to guarantee the correctness of the product have to be explored. One basic method used to test the correctness of a chip is to find a small set of input patterns (a set of tests vectors) which test all logical paths through the circuit. These input patterns are created using test vector generation algorithms based on different fault models. Other methods include adding extra outputs to sample the data within the circuit or adding special circuitry to the chip which tests if it functions correctly (built-in self-test).

Within the Computer Architecture and Digital Systems (CARDIT) laboratory of the Information Technology and Systems (ITS) faculty at the Delft University of Technology research is done into the field of test algorithm development. For this purpose an integrated software frame-

work called DAT (Delft Advanced Test generator) has been developed. This framework accepts a netlist circuit description of a digital circuit as input, and then automatically generates a set of test vectors based on the chosen fault model. Another feature being developed by the CARDIT department is the automatic generation of built-in self-test circuitry.

1.2 The purpose of an Automatic Schematic Generator

The netlist circuit description, which is used as input for the DAT framework, is in turn itself machine generated. The netlist file is essentially a summation of the components used in the circuit and the interconnections between the components. These days almost all circuitry is designed in a hardware description language, such as VHDL. This allows the designer to specify the function of the schematic without having to think about all the details. More complex designs can be created in the same amount of development time. However, for large circuits the netlist generated from the hardware description is completely incomprehensible for humans.

For test algorithm designers, the most comprehensible design representation is the schematic diagram. The schematic diagram can clearly show the effect an input pattern has on different logical gates, and help visualize which parts of the circuit are activated by a particular test vector. Since manually transforming a netlist description into a schematic diagram is very time consuming, even for small circuits, an automatic schematic generator has been added to the DAT system. This has the following advantages:

- Help the students and staff developing test algorithms to visualize areas of the schematic where faults are difficult or impossible to detect.
- Help the students and staff developing test algorithms to debug their software by visualizing the circuit when their algorithms have unexpected results.
- Show structural dependencies within a digital circuits, such as Fanout Free Regions (FFR's).
- Enable demonstration of test algorithms by visualizing the effect of signal assignments on the circuit.

1.3 Introduction to the existing Automatic Schematic Generator

Before this assignment was started, the DAT framework already contained a rough version of an Automatic Schematic Generator (ASG) (see [Sab94] for the details). It contained a number of bugs and shortcomings, as will be explained in later chapters. It also lacked a number of features such as displaying structural dependencies, which are useful for the reasons mentioned earlier. The ASG used very simple algorithms to solve various subproblems of an ASG. However, it was sufficient to demonstrate the usefulness of adding an ASG to the DAT framework. Therefore, it was decided that the current solution should be used as a base for an improved ASG.

1.4 Task Assignment Overview

Task description: *Schematic Generation*

Research will be carried out as part of the DAT test generation platform, using the programming language C++.

1. Literature study:
 - Carry out a study of existing literature of schematic generation.
 - Study the technical report (see [Sab94]) of the existing schematic generator.
 - Study existing public domain software which can be used for the assignment.
2. Fix bugs and shortcomings in the current ASG implementation.
 - Find and remove errors in the ASG output.
 - Improve and optimize the existing ASG algorithms.
3. Extend and improve current options:
 - Algorithm related: implement new algorithms to draw the schematic diagram.
 - Schematic related: optionally switching the sequence of the circuit's input pins and modules' input pins.
 - Animation related: add options to highlight input and output cones, FFR's and regions

1.5 Report Overview

This section will conclude the introduction by giving an overview of the report's structure. In order to understand the problems and terminology of designing an Automatic Schematic Generator (ASG), Chapter 2 will introduce the used terminology, describe the problem in details, and show the general solution to the ASG problem. Chapter 3 shows the results of the literature study. The original implementation of the ASG, which is the basis of the current implementation, will be discussed in Chapter 4. Chapter 5 presents the various bug fixes, algorithm improvements and functionality extensions which have been made to the original implementation. Chapter 6 presents the conclusions and recommendations.

The Automatic Schematic Generation Problem

2

This chapter gives an overview of the general solution to the Automatic Schematic Generator (ASG) problem. First, an introduction to the ASG terminology will be given in Section 2.1. The ASG's input, the logic gates and their interconnections, will be described in Section 2.2. The ASG's output, the schematic diagram, will be described in Section 2.3. The general ASG problem decomposition, as encountered in most reviewed papers, and the base of the existing DAT-ASG implementation, is discussed in Section 2.4. The problem is decomposed into four subproblems which are discussed in the following sub-sections. This chapter concludes with an overview of the general ASG solution in section 2.5.

2.1 General ASG Terminology

This section gives a brief overview of some of the terminology which is used throughout this report. It does not attempt to give a complete overview, but only the most basic terms.

- Schematic Diagram - the graphical representation of a circuit.
- ASG - Automatic Schematic Generator. A tool to automatically draw schematic diagrams from a circuit description. The topic of this research.
- Module - General term for a building block in the circuit. In this case mostly Boolean gates such as AND, OR, NOT. Other modules are circuit inputs and outputs, and fanout points.
- Connection - General term for the physical connection between two modules. On the schematic diagram, these are the signal lines between the modules.

- Placement - the ASG subproblem of finding the position of a module on the schematic diagram.
- Routing - the ASG subproblem of finding the position of a connection on the schematic diagram.
- Column and Row Position - the modules in the circuit are initially placed in columns of modules (see figure 2.4). These columns are numbered from left to right. The sequence of the module within its column is referred to as the row position. Each pair of (column, row) can be seen as a position on a logical grid, and cannot be occupied by more than one module.
- Channel - the area between two consecutive columns of modules where the connections are drawn (see figure 2.5(a)). The channels are numbered from left to right.
- Track - on the schematic diagram connections consist of horizontal and vertical line segments. As long as connections have horizontal line segments only, the routing problem is trivial since there can be no overlap of connections. However when a connection's source and destination module have different row positions, a vertical line segment is needed. A track is a vertical line in a channel where the vertical line segment of a connection can be routed (see figure 2.5(b)). If the vertical line segments of different connections do not overlap each other, they can be assigned to the same track. Per channel the tracks are numbered from left to right. By assigning a track number to each connection which needs a vertical line segment for routing, the ASG program can keep track of the use of channel space and prevent overlap of different connections.
- Passthrough - a connection between two modules in non-consecutive columns (see figure 2.1). For each column the connection skips, one passthrough is needed.
- Module's predecessors and successors - modules connected to the inputs and output of a module.

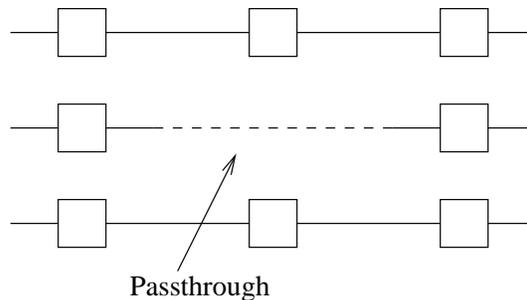


Figure 2.1: Example of a Passthrough.

2.2 The Netlist Description

With the increase in size and complexity of modern digital circuits, the primary hardware design method has shifted from hand drawn schematics to the functional description of the circuit in a hardware description language (such as VHDL). These descriptions are then translated by a circuit compiler into what is called a netlist.

The netlist is a list of basic logic gates which form the digital circuit. These basic gates perform the following basic logic functions:

- AND, NAND, XAND
- OR, NOR, XOR
- NOT (Invert)

With these logic functions, every boolean combinational circuit can be described. The netlist does not show any technology-dependent information such as timing, and is essentially a list of all basic logic gates used to implement the circuit's function and the interconnections between the gates, as described by its creator in the HDL. This netlist in turn is translated into a chip layout which is used for production.

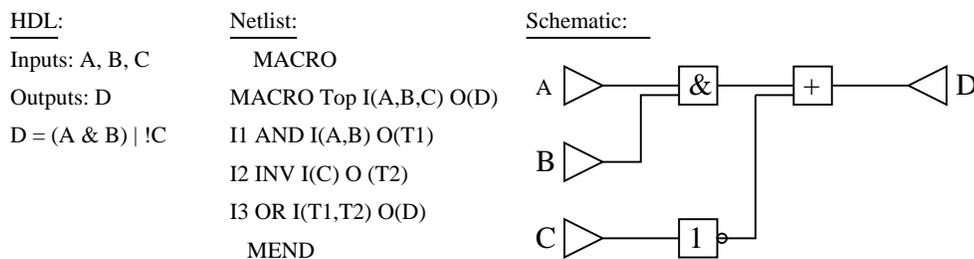


Figure 2.2: Example of a Netlist and its corresponding Schematic.

The advantage of the netlist is that it is highly portable. It only lists logical components and their interconnections, and no physical components. Therefore, it is technology independent. This also means that the amount of information that is stored in the netlist is limited, and all information can be compactly stored in a textual form.

The disadvantage of the netlist is that although it shows the used components and their connections, it is almost impossible for humans to understand the functional structure of the circuit. Only for very small circuits can the designer, or anyone else interested in the circuit structure, such as the test algorithm developer, transform the netlist into a schematic. This implies the need for automatic netlist-to-schematic translation: an Automatic Schematic Generator (ASG).

Figure 2.2 gives an example of a HDL, the resulting netlist and its corresponding schematic diagram. Even for such a simple case, the function and structure of the schematic are not immediately clear to the observer by just looking at the netlist description. Once the number of gates increases past a few tens, it becomes increasingly difficult to manually reconstruct a circuit's schematic or function based on the netlist description only. Modern digital circuits can include as many as tens of thousands of gates, or even millions.

2.3 The Schematic Diagram

Schematic diagrams were once the principal design medium for (digital) circuits, providing a clear and graphical representation of a circuit. With the current complexity and size of digital circuits, it has become impractical to design circuits with schematics. However, the schematic is still useful to provide additional insight. Especially in the field of test algorithm development, schematics are a useful representation to show the exact effect a test vector has on the circuit.

A complicating factor in the design of an automatic netlist-to-schematic generator is the fact that the netlist has no obvious resemblance to the original design in the hardware description language. Most hardware description languages allow the hierarchical build-up of modules. For example, a single circuit designed in a hardware description language can consist of several submodules, which in turn also consist of submodules, etc. When such a final module is translated into a flattened description, it will just be a long list of logical gates and their interconnections.

If such a netlist were to be transformed into a schematic diagram by a number of different human designers, each of them would come up with their own schematic. If all of them were then asked to select the best schematic diagram, they would never come to a unanimous decision. There is no single quantitative method to determine which is the best schematic: each designer will have his or her own personal preferences regarding the final layout of the schematic.

However, by observing human made schematics, a number of general characteristics can be determined:

- A schematic consists of two basic items: modules (in this case logic gates) and connections between the modules.
- All connections consist of horizontal and vertical line segments only (forming a Manhattan style graph).
- Signals flow as much as possible in one general direction: usually from left to right.
- The number of connections crossing each other is minimized to make the connections easier to follow with the eye.
- The number of bends in the connections is minimized to make the connections easier to follow with the eye.
- The length of the connection lines is minimized to make the connections easier to follow with the eye.
- The circuits' inputs and outputs are drawn at opposite edges of the schematic diagram. This of course forces a general directions in the signal flow.
- The modules are usually drawn underneath each other in columns.

For the automatic netlist-to-schematic generator, these characteristics result in the following conditions, which are also shown in figure 2.3:

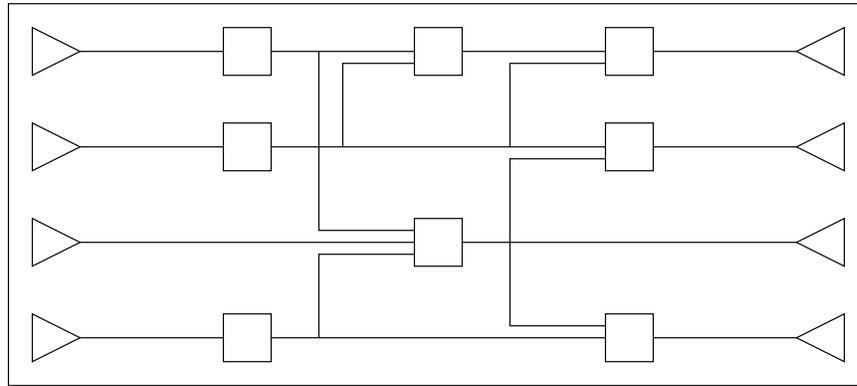


Figure 2.3: Example of a Schematic Diagram.

- The schematic diagram occupies a rectangular area between the input and output terminals.
- All inputs are drawn at the leftmost side of the schematic.
- All outputs are drawn at the rightmost side of the schematic.
- All signal flow is from left to right only.
- The schematic diagram consists of columns of modules alternated by channels for routing.
- All connections between modules consist of horizontal and vertical line segments only (Manhattan graph style).
- No module overlaps with another module or connection, and vice versa.
- All feedback loops are cut and translated into so-called internal primary inputs and internal primary outputs. These are then treated as normal input or output terminals. This is done for ease of implementation.

2.4 General Problem Decomposition

The existing DAT-ASG, as well as most reviewed literature, use a general problem decomposition strategy. This general problem decomposition is explained in this section and the following sub-sections.

The first step in decomposing the ASG problem is to distinguish between module placement and connection routing. Module placement involves calculating the exact position of each module on the drawing surface. Connection routing involves calculating the exact location of each connection on the drawing surface.

The second step in decomposing this problem is to distinguish between logical placement and routing, and geometrical placement and routing. Logical placement involves assigning

positions to modules and connections on an abstract (non-physical) drawing surface. Geometrical placement and routing involves assigning exact coordinates to modules and connections.

These four basic problem decomposition steps will be discussed in-depth in the following sub-sections. The final phase, of course, is to display the results on either the computer terminal, or write them to a file for later printing and processing.

2.4.1 Logical Module Placement

The first step in decomposing the ASG problem is calculating the logical placement of modules. Logical placement involves assigning a row and column position on a logical grid to each module in the circuit. The logical grid is illustrated in figure 2.4. Each position (a square) on the logical grid can be occupied by one module. By assigning each module to a grid position the placement problem can be solved independent of the final placement of modules on the drawing surface of the schematic.

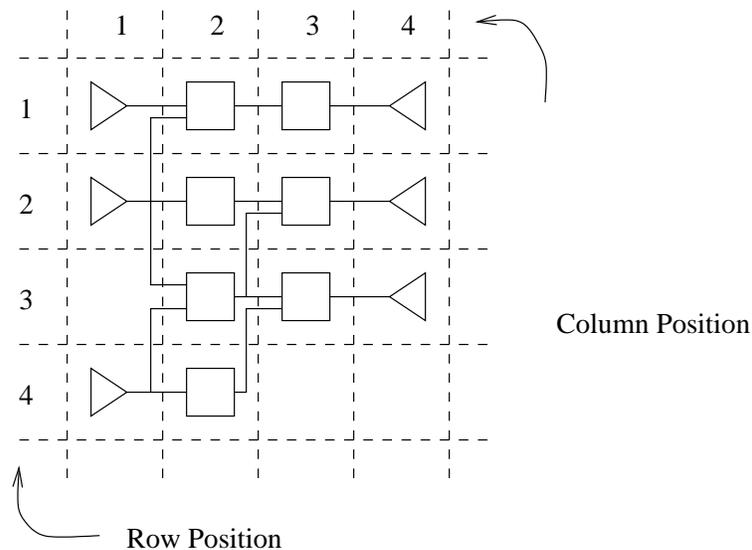


Figure 2.4: The logical Grid.

Once the logical placement phase is completed, the connections between the different columns of modules can be routed. This step is called logical routing, and treated in the next section.

The general solution for the logical placement problem is to first calculate the depth level (distance from the input pins in logic gates) of each module in the circuit. This is usually done recursively. First, the circuit inputs are all assigned a depth level of zero. After this, if all predecessors of a modules have a level assigned to them, the module is assigned a level equal to the maximum level of it's predecessors plus one, or:

$$Level = \max(\text{level of each predecessor}) + 1$$

In order for this to work properly, all feedback loops have to be removed. In DAT this has already been done by cutting the loops around the latches. The input signal of the latch is treated as a circuit output, and called an internal output. The latch's output is treated as a circuit input, and called an internal input. For ASG purposes internal inputs and outputs are processed the same as normal circuit inputs and outputs.

After depth levels have been assigned to each module in the circuit, these depth levels are then used to calculate the column position on the logical grid. This is usually done by setting the column position equal to the depth level. However, if a module's depth level equals i , and the minimum depth level of its successors equals j , it can be assigned a column anywhere in the $[i, j-1]$ interval. As an example, if the predecessors of module M have depth levels 1, 4 and 7, and the successors have depth level 11 and 12, module M can be placed anywhere in the interval $[8, 10]$. The exception to this are the circuit's inputs and outputs, which are always assigned to the first and last column of the grid which keeps them in a straight vertical line. This also simplifies the design of the ASG.

Once each module is assigned to a column on the logical grid, the row position can be calculated. There are many different methods to calculate the row position, based on the optimization criteria and the choice of heuristic. Common optimization criteria include the minimization of signal line crossings, the number of signal line bends, and the total signal line length. These criteria show one of the major problems in ASG design: there is no 'best' solution. By minimizing the number of crossovers, you might increase the number of signal line bends and the total wiring required. And even if you focus on one criterion only, the problem is already NP hard. If for example a column contains N modules, there are $N!$ possible sequences of modules, each of which will have to be evaluated to find the best solution according to your optimization criterion.

2.4.2 Logical Connection Routing

The second step in the basic problem decomposition of the ASG problem is calculating the logical routing. This step will conclude the logical phase, after which the geometrical phase can start. The geometrical phase calculates the exact position of each module and connection on the drawing surface of the schematic.

Logical routing involves assigning a track number to each connection in the channel where the routing takes place. The channel is the space between two consecutive columns of modules, as shown by figure 2.5(a). Tracks (see figure 2.5(b)) are vertical lines in a channel to which connections are assigned. These vertical line segments may not overlap each other. The logical routing problem consists of assigning a track number to each connection needing a vertical line segment for routing. This means that the connection's source and destination module have different row positions on the logical grid. The tracks are numbered from left to right.

One obvious restraint with logical routing is that track assignment must be done in such a way that connections do not overlap each other. Connection overlap means that the line segment used to route different connections partly overlap each other. When this happens it becomes very difficult to follow connections from the source to their destination module. Preventing overlap

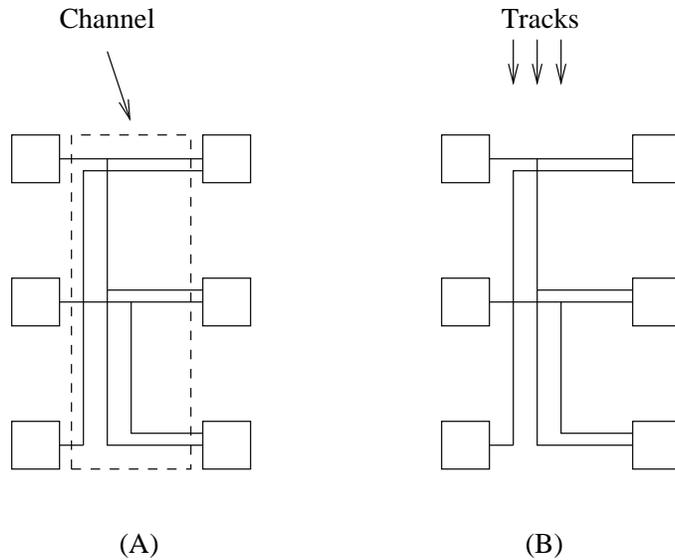


Figure 2.5: Example of a Channel and some Tracks.

places restrictions on the sequence in which connections can be assigned to tracks. There are two types of possible overlap: vertical and horizontal. Vertical overlap means that the vertical line segments of two different connections overlap each other. Horizontal overlap means that the horizontal line segments of two different connections overlap each other. The two types of overlap are illustrated in figure 2.6. In this figure the signal lines connect modules with the same number. Since the line segments (partly) overlap each other, this has become unclear. Vertical overlap (figure 2.6(a)) happens when two connections, whose vertical line segments overlap each other partly, are assigned to the same track. Horizontal overlap (figure 2.6(b)) happens when connection 2's vertical line segment end at the Y coordinate where connection 1's vertical line segment starts, and connection 2 is assigned to a lower track number. Or in terms of the logical grid: when connection 2's destination modules has the same logical row position as connection 1's source, and connection 2 is assigned to a lower track number then connection 1.

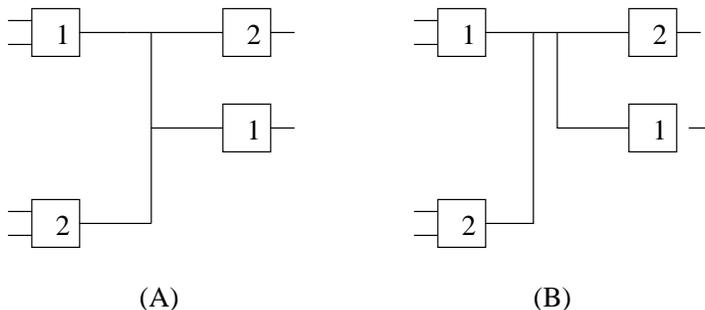


Figure 2.6: Examples of overlap, (a) vertical and (b) horizontal overlap.

The optimization criteria in this phase are to minimize the number of signal line crossovers

and the number of tracks needed for routing. However, by minimizing the number of crossovers, the number of required tracks might increase, and vice versa. Also, even when optimizing logical routing for only one of these criteria, the problem is probably NP hard. Although no exact proof of the complexity of the logical routing problem has been found in papers, the solution space will be too large for exhaustive search methods. To find for example the optimal logical routing based on minimizing the number of signal line crossovers, all possible combinations of connections sharing tracks will have to be considered and evaluated. Even if every connection was assigned to its own track, there are $N!$ possible sequences of tracks which will have to be evaluated for validity and the number of signal line crossovers generated.

2.4.3 Geometrical Module Placement

The third step in the basic problem decomposition of the ASG problem is calculating the exact coordinates of each module on the drawing surface of the schematic. This is usually done per column of modules, and from left to right, starting with the circuit inputs. Alternative strategies include starting with the column containing the most modules.

The exact position of a module can be calculated in a number of different ways, based on the row and column positions of the module on the logical grid, the position of the previous column of modules, the number of required tracks between the previous column of modules and the current column, and the exact position of the predecessor modules. For DAT, we assume a direct mapping between the logical grid positions and the final geographical coordinates on the schematic. This simplifies the geometrical placement problem considerably. This is further explained in Section 4.6.2.

2.4.4 Geometrical Connection Routing

The fourth and final step in the basic problem decomposition of the ASG problem is calculating the exact coordinates of each connection on the drawing surface of the schematic diagram. This involves calculating the begin and end coordinates of the different line segments forming a single connection. In DAT's ASG implementation this can be done in an arbitrary sequence, since by now the source and destination location of each connection are known, as well as the track number the connection is assigned to. Except for fanout stems and branches this step is fairly straightforward. The details are explained in section 4.6.3.

2.5 General ASG solution Overview

This section giving an overview of the general ASG solution as discussed in the previous sections.

General ASG Solution:

1. Logical Module Placement:

- (a) Remove feedback loops.
- (b) Assign depth level to all modules.
- (c) Assign each modules to a column of the logical grid (column assignment).

- (d) Determine for each column of the logical grid the sequence of the modules assigned to it (row assignment)
2. Logical Connection Routing:
 - (a) assign a track number to each connection.
3. Geometrical Module Placement:
 - (a) Calculate the exact coordinates on the schematic's drawing surface of each module in the circuit.
4. Geometrical Connection Routing:
 - (a) Calculate the exact position of each connection in the schematic diagram.
5. Display the results:
 - (a) Draw all modules on their calculated position.
 - (b) Draw all connections, using the calculated positions of their line segments.
 - (c) Display the drawing on-screen, or save to a file.

Survey of existing Literature

3

This chapter reviews 13 papers which appeared useful as research material for the Automatic Schematic Generator (ASG) and related problems. The papers are reviewed in chronological order. Each paper has its own section. The papers will be reviewed in terms of the general ASG solution presented in Section 2.5.

General ASG Solutions Ahlstrom (Section 3.3) tries to solve the ASG problem using an expert system. The paper describes an explorative placement and routing method. It places the modules one at a time, using a set of placement and routing rules which are not described.

Kumar (Section 3.7) and Chun (Section 3.9) focuss on the optimization criteria of the ASG process and discuss ASG solutions based on the general solution method without discussing the details of their solution methods.

Stok (Section 3.10) and Venkatarama (Section 3.8) describe a placement and routing method, which partitions the circuit into a number of strings. These strings, a number of elements connected to each other, are then treated as individual circuits. After the schematic diagrams of individual strings are calculated, they are connected back together to form the whole schematic.

Placement and Routing methods Jehng (Section 3.13) presents a logical placement and a logical routing method. For logical placement it uses a method called Bubble Sort. The Bubble Sort method sorts the element columns based on the average position of an element's predecessors. For logical routing it uses a method called Modified Left Edge Routing, base on a VLSI layout algorithm. This method first sorts the connections according to the row posi-

tion of the source elements. It then assigns connections to tracks using a modified first fit method.

Lee (Section 3.11) presents logical placement method. A simulated annealing procedure is used to calculate module positions on a logic grid. The paper also describes a procedure to detect tree structures in the schematic. The tree detection algorithms are explained in general terms only.

May (Sections 3.4 and 3.5) and Majewski (Section 3.6) present a logical placement method. A matrix based solution is used to calculate the module positions on a logic grid. Using a matrix containing connectivity information, a placement method was developed which focuses on minimizing the number of signal line crossovers.

Swinkels (Section 3.12) presents a logical placement method. The paper described a knowledge based system (KBS) approach to the ASG problem. It explains a useful grid assignment method, called Barycentric Sort, which uses the average row position of an element's predecessors to calculate its row position. It also explains the use of passthrough or link elements to deal with connections between modules in non-consecutive columns.

Brennan (Section 3.1) presents a routing method. It calculates the line segments needed to route connections, by dividing the drawing surface in a collection of horizontal and vertical line segments. By assigning connections to line segments the router keeps track of the segments which are used already and those that are still available.

Yoshimura (Section 3.2) presents a logical routing method. It explains a graph based approach to assign connections to tracks. By portraying the routing restrictions, due to overlapping connections, in a graph, a solution which minimizes the number of tracks was found.

3.1 "An Algorithm for Automatic Line Routing"

See [Bre75] for the complete version of this paper.

The algorithm described in this paper has been developed for the Bell System Schematic Drawing. Its primary goal is to automatically generate connection routings.

This paper focuses mainly on the routing aspect of ASG. It assumes that all modules have already been placed on a grid layout. It only needs a list of connections between modules to generate the routings. The schematic drawing is placed on a grid system, which exists of a collection of horizontal and vertical line segments. A list of used segments is stored in memory.

The algorithm searches the connection list for common aspects. For example, if a signal line branches in two parts, both of which have different destinations in the next column, these destinations share a common X coordinate. This means that both connections share a horizontal line segment.

If multiple solutions for a connection exist, the algorithm uses weighing criteria such as the number of different line segments (i.e. the number of bends in the signal line) and the amount of crossovers.

If a final solution for a signal line is reached all grid segments used by the signal line are marked, and the next signal line is resolved

The advantage of this routing method is its general usage. It makes no difference between feedback signal lines, branches, and normal straight connections. It just analyses a list of connections, sorts this list based on common aspects, and calculates the line segments needed to route connections.

Disadvantages of the algorithm described in this paper are:

- The algorithm has no guaranteed solution for the routing problem. It has no backtracking, so if it encounters a connection that cannot be realized due to earlier made connections, the algorithm stops.
- The algorithm has no signal flow restrictions. It may use routing with signal flow from left to right, as well as right to left. This makes the resulting routing more difficult to read.
- It algorithm does not estimate or calculate the amount of empty space around the modules which is needed for routing. The algorithm simply assumes a fixed area between the modules. It does not specify what to do when the amount of space is insufficient for routing.

3.2 "Efficient Algorithms for Channel Routing"

See [TY82] for the complete version of this paper.

This paper describes two new algorithms used for channel routing in VLSI layout design. The channel routing problem in VLSI layout design is a generalized case of the channel routing in the ASG problem. However, note that the objective within VLSI applications is to minimize the area needed for routing, as opposed to ASG applications where the main concern is the readability of the schematic.

Since this paper focuses on VLSI layout applications for the proposed algorithms, only relevant sections will be treated here.

In this paper the channel is represented as two horizontal rows of pins. The channel is the space between the modules, where the connections are routed. Usually, the channel is represented as the space between two columns of modules, with the signal flow going from the left column's output pins to the right column's input pins, as is illustrated by figure 2.5(a). The channel as used in this paper is illustrated in figure 3.1(a). Each pin has an integer value assigned to it. These are the numbers 1 to 10 at the top of the figure. The connections are drawn between the two rows of pins, and numbered 1 to 7. In memory, the channel is represented by two arrays. These are illustrated in figure 3.1(b). The top row of terminals in the channel is the top row of numbers in the box, and the bottom row of terminals in the channel corresponds with the bottom row of numbers in the box. The sequence of the terminals corresponds with the sequence of the numbers, and the numbers correspond with the connection number assigned to that terminal. If a terminal position has no connection attached to it, it is assigned a zero value. Terminals with corresponding numbers are connections between those terminals. Each

connection may only have one horizontal and two vertical line segments.

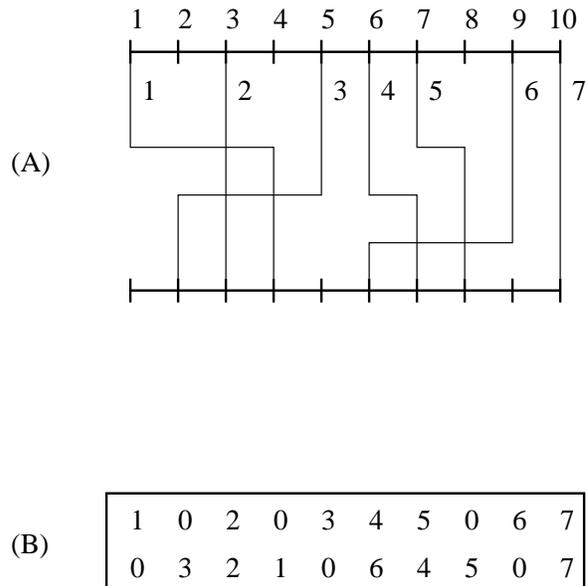


Figure 3.1: The Channel as described in [TY82].

To solve the channel routing problems, two graphs are constructed. These are:

1. Vertical Constraint Graph (VCG)
2. Horizontal constraint Graph (HCG)

The Vertical Constraint Graph (VCG) is a directed graph (see figure 3.2(b)). Each node represents a connection in the channel. Each edge between two nodes indicates the connection represented by the first node must be placed in a track (here: horizontal line) above the second to prevent overlap.

Figure 3.2(b) shows the VCG of a sample channel. The edges in the VCG indicate that connection 1 must be assigned to a horizontal line above connection 3, and connection 3 must be assigned to a horizontal line above connection 4. If for instance connection 3 is assigned to a horizontal line above connection 1, connection 1 and 3 would have an overlapping vertical line segment directly below terminal 1.

The Horizontal Constraint Graph (HCG) is a non-directed graph (see 3.2(C)). Each node represents a connection in the channel. Each edge between two nodes indicates that the connections represented by the node would overlap if they are drawn on the same horizontal line (track).

Figure 3.2(c) shows the HCG of a sample channel. The edges between node 1 and the nodes 2, 3 and 4 indicate that connection 1 cannot be assigned to the same vertical line as connection 2, 3 or 4. Likewise it indicates that connection 2 cannot be assigned to the same horizontal line as connection 1 and 3, and connection 4 cannot be assigned to the same horizontal line as

connection 1 and 3.

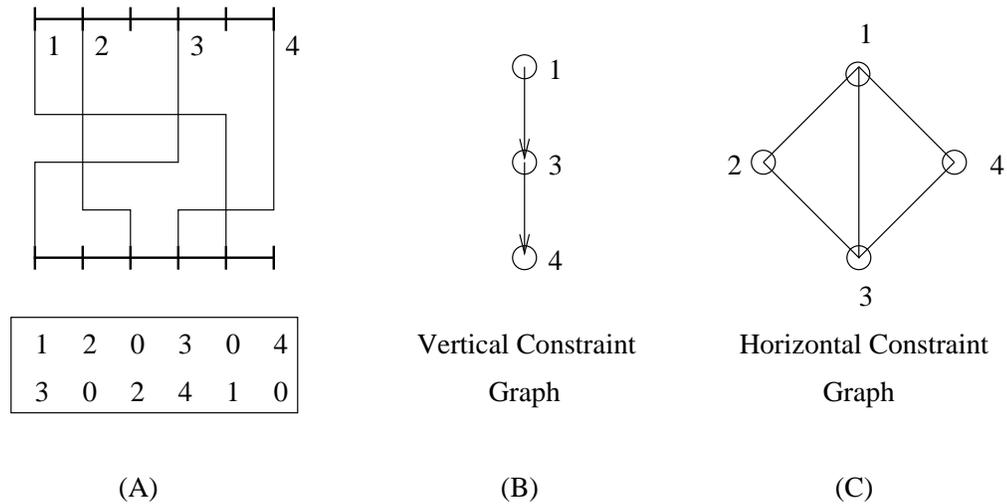


Figure 3.2: Example of a channel (a) and its corresponding VCG (b) and HCG (c).

The routing problem discussed in this paper consists of finding the minimal number of tracks required for routing. The number of tracks needed for routing can be deduced from the VCG and HCG. First the VCG graph is leveled. The nodes in the VCG which have no edges leading to them are assigned a depth level of one. From there the depth level of all nodes is calculated based on their input nodes' depth level. This is demonstrated in figure 3.3. Each node is assigned a depth level of:

$$DepthLevel = \text{Max}(\text{depthlevel of all source nodes}) + 1$$

The connections represented by nodes with the same depth level are then checked in the HCG. If they have no horizontal overlap restrictions, they can all be assigned to the same track. If the HCG shows that connections with the same depth level in the VCG will overlap each other, each of the overlapping connection needs its own track. The total number of tracks is found by checking all depth levels of the VCG and calculating the number of required tracks per depth level.

If however the VCG contains loops, the situation get complicated. A loop in the VCG indicates that routing the connections forming the loop, is impossible using a single horizontal line segment (in this paper, the tracks are horizontal). Or, in other words, one of the connections represented by the graph's loop needs to be routed over two tracks, using five line segments. This is demonstrated by figure 3.4, where after routing connection 1 and 2 it is not possible to route connection 3, without having overlapping connections or using two horizontal line segments.

The VCG and HCG graphs completely characterize the channel routing problem. The objective in this paper is to find an optimal ordering of connections such that:

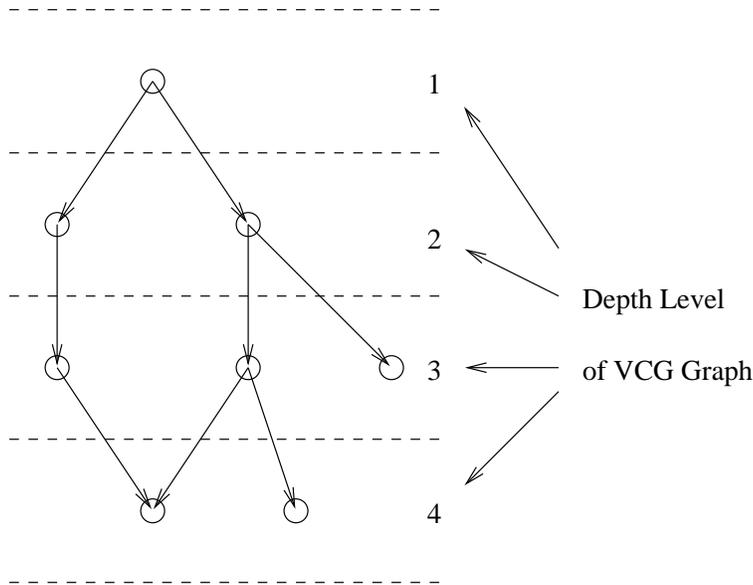


Figure 3.3: The levels in a VCG.

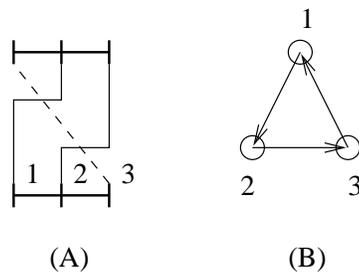


Figure 3.4: Example of a loop in the VCG graph, (a) a channel and (b) the matching VCG graph.

1. The constraints in the Vertical Constraint Graph are satisfied
2. The constraints in the Horizontal Constraint Graph are satisfied
3. The number of tracks needed for routing is minimized

Note that if all connections are allowed to have more than one horizontal line segment (or: more than three line segments), the number of tracks needed for routing may be further reduced (dogleg problem). This however clearly reduces the readability of the schematic, since three segment lines are easier to follow with the eye than five segment lines. For chip layout channel routing, where reducing the number of tracks and thus reducing the area needed for routing is the most important optimization criterion, this is not a problem.

The first algorithm to solve the channel routing problem is based on the merging of connections. It states that connections i and j may be represented by a single one if:

1. There is no edge between i and j in the HCG

2. There is no directed edge between i and j in the VCG

Or in other words: the two connections i and j can be placed on the same horizontal track.

Merging i and j has the following consequences for the VCG and HCG:

1. In the VCG, node i and j are replaced by a single node $i&j$. All edges leading to either i or j now lead to $i&j$. All edges leaving i or j leave $i&j$.
2. In the HCG, the nodes representing i and j are also replaced by a single node $i&j$, and all edges connected to either i or j are now connected to $i&j$.

The process of merging is illustrated in figure 3.5. If connections 1, 2 and 3 have no horizontal constraints, they can be replaced by a single node in the VCG. The same applies to connections 4 and 5. This of course assumes that there are no horizontal constraints stopping these merges.

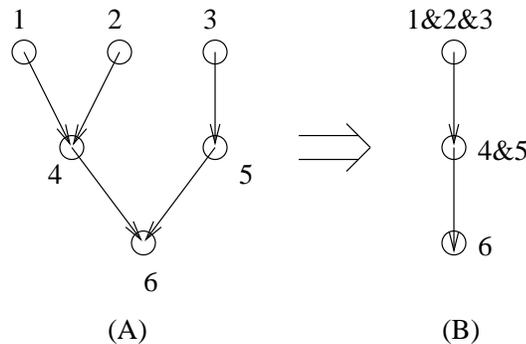


Figure 3.5: Example of Net Merging using the VCG, (a) before and (b) after.

The second algorithm is an extended version of the first. A third graph, called the merging graph, is constructed. Each node represents a connection, and each edge between connection node a and b represents that connection a and b can be merged according to the conditions specified in algorithm 1. Using the merging graph, the HCG and VCG are examined to see which merging would reduce the number of levels in the VCG (and thus probably the number of required tracks).

Although the proposed algorithms are designed for VLSI layout design applications, they can also be used for ASG implementations. However, they produce very irregular routing, since they only focus on reducing the number of required tracks. Therefore, these algorithms should be at most be optionally available within the schematic generator. The graph approach however seems useful for further research into suitable track assignment algorithms.

3.3 "HAL: A heuristic Approach to Schematic Generation"

See [MA83] for the complete version of this paper.

The drawing of schematic diagrams requires a considerable amount of expertise on behalf of the creator if the resulting schematic is to be aesthetically pleasant for human viewers. The creation of a schematic is partly an artistic task. Since it is difficult to specify algorithms for artistic tasks, this paper describes an expert system based approach to the schematic generation problem.

The first and second part of the paper describe the interview method and the software / hardware platform for the implementation.

The third section describes the placement mechanism. For this it uses a place and explore algorithm. The system places a module, examines which modules are connected to its input and output pins, and then selects one of these as the next module under investigation (based on its placement rules). The exact rules for placement and position calculation are not mentioned in the paper.

The fourth section involves the routing rules. Except for stating the presence of an overlap detection mechanism, no implementation details or heuristics are mentioned.

The system demonstrates the application of rule-based expert system technology to the ASG problem. It is however very limited in what it does. Considering the programming and knowledge gathering effort that was put in the design of HAL, the paper concludes that algorithm based solutions seem more appropriate for the ASG problem than a rule-based system.

3.4 "Placement and Routing for Logic Schematics"

See [MM83] for the complete version of this paper.

This paper describes the automatic synthesis of logic schematics. It's main focus is on logical placement (part of the placement problem), optimized for a minimal amount of signal line crossovers. It describes an extensive algorithm, based on matrix calculations and graph theories.

The row placement optimization problem consists of finding a sequence of modules within each column which results in the minimal amount of signal crossover in connections between the columns.

This paper describes a recursive algorithm to find the optimal row ordering which results in the minimal amount of crossovers. Each iteration has a computational complexity of $O(C * N_{max}^2)$, where N_{max} is the maximum amount of modules within a single column, and C the number of columns. This is a monotonous algorithm, which means that the solution of the current iteration at worst equals that of the previous iteration.

The only item discussed at length in this paper is the row assignment problem, part of the placement problem, which in turn is part of the ASG problem. The algorithm discussed

in the paper claims to always find the minimal amount of signal crossover. However, it still takes a considerable number of iterations (and time) to calculate the optimal row sequence of the modules, especially for schematics with a large number of modules and connections. A second disadvantage is that the matrix calculations required for this algorithm are difficult to implement. Also, there are more optimization criteria, apart from crossover minimization, that can be used for the row assignment. It is however still an interesting option for further research. When the DAT-ASG has enough time to calculate a schematic based on the absolute minimum of line intersections, this algorithm can prove to be useful.

3.5 "Computer-Generated Multi-Row Schematics

See [May85] for the complete version of this paper.

This paper describes an algorithmic solution to the ASG problem. It uses a graph theoretic approach for the logical placement of modules. The geometrical placement aims at improving the aesthetics of the drawing. A modified channel routing algorithm is used for routing (no details are mentioned in the paper). The authors have implemented the algorithms.

Prior to placement, link nodes are included in the schematic. Link nodes are pseudo modules with one input and one output terminal, which have no logical function. These nodes are used when a source module's output terminal is connected to a destination module's input terminal in a non-consecutive column. Adding link nodes between the source and destination modules ensures that enough free space is reserved for routing.

This paper mainly focuses on the logical placement problem, part of the module placement problem. A graph is constructed from the current column and the column to its right. The modules in the columns form the vertices in the graph. Two optimization methods are discussed to reorder the modules.

- The first optimization method discussed uses the number of line crossovers in the graph as a criterion. The graph, representing two consecutive columns, is used to construct a matrix which stores all connectivity information for the two columns. Using an iterative algorithm, an optimal solution is found. The algorithm has a computational complexity of $O(C * N_{max}^4)$ per iteration, where C is the number of columns of the schematic, and N_{max} the maximum number of elements present in a single column.
- The second optimization method discussed uses the total length of all the edges of the graph (representing the two consecutive columns) as optimization criterion. Using an iterative algorithm, the optimal solution is found. The algorithm has a computational complexity of $O(P * N_{max}^2)$ per iteration.

The authors also show an alternative geometrical placement method. The modules of the first pair of consecutive columns are assigned a physical position such that the maximum number of horizontal lines that make up the circuit area are used. The same is done for the second pair of columns (columns two and three), taking the coordinates of the first pair into consideration.

Reaching the last segment, final positions are calculated. The procedure now runs in reverse from the last to the first column, updating the coordinates in segment $i-1$ according to the positions in segment i , while preserving the number of horizontal lines. This method can be used as the basis for a recursive geometrical placement algorithm. Each iteration, the circuit will be scanned from left to right, assigning positions which are relative to the previous column. After reaching the rightmost column, final position assignments are made from right to left, based on the relative positions which have been calculated.

Although this method requires more computational time (the exact costs of each iteration are not mentioned in the paper) than straightforward geometrical placement, temporarily results from early iteration steps can already be displayed while the ASG program calculates better alternatives. However, the DAT-ASG implementation uses a fixed area of the schematic for each logical grid position. This was done to simplify the geometrical routing problem. Due to time restraints this has been left in tact, since it is an assumption of the DAT-ASG which is used throughout the implementation.

The usage of link modules, which are the same as passthroughs, is useful to solve the problem of multi-column routing and has been implemented in the DAT-ASG implementation. In this paper the link modules are sorted per column in the same manner as normal modules, as opposed to the DAT-ASG approach where passthroughs can only be placed at the same row position as their source module. The disadvantage of this method is illustrated in 3.6. When a connection consists of a series of link modules (passthroughs), and each link module is assigned to a column independently of the other link modules, they could all end up in different logical rows. This would reduce the readability, so the DAT-ASG approach of placing all passthrough needed for a single connection at the same logical row position is better.

The optimization methods discussed in the paper seem memory and time consuming; each column that is optimized requires the initialization of a complex graph data structure, and a number of iterations. The crossover minimization is a popular optimization method for placement. Minimizing the total length of the signal lines between two consecutive columns (the second optimization method) is an approach which is already present in the DAT-ASG implementation as a consequence of the logical placement algorithms.

The geometrical placement method used in this paper looks interesting. Although it inserts extra empty space in the circuit, the total circuit area (the area enclosed by a rectangular box around the total schematic) stays the same, while improving the clarity of the schematic. The paper however speak in very general terms about the algorithms used. The routing algorithm which is used is not mentioned.

3.6 "Autodraft: Automatic Synthesis of Circuit Schematics"

See [MMA86] for the complete version of this paper.

This paper describes the Autodraft system, a schematic generator for use with a prototype silicon compiler, called Autocircuit, under development at General Motors Research Laboratories. Autocircuit uses gate level descriptions for high level building blocks or modules,

which are stored in a database and can be used in other designs. Autodraft is used to generate readable schematics of each building block.

The Autodraft system follows the general approach for the ASG problem as explained in Chapter 2. It used a logical grid to abstract placement, and assigns a row and column position to each module present in the schematic. The column assignment algorithm used in Autodraft is a non-standard approach. First, an adjacency (connectivity) matrix A is formed, whose elements are assigned binary values as follows:

$$A_{ij} = \begin{cases} 1 & \text{output of module } j \text{ drives input of module } i; \\ 0 & \text{otherwise.} \end{cases}$$

A column vector F_k is used to indicate for each module if it is present at the k^{th} level of depth. The elements of F_k are defined as:

$$F_{k_j} = \begin{cases} 1 & \text{module } j \text{ at } k^{th} \text{ level in graph;} \\ 0 & \text{otherwise.} \end{cases}$$

The descendant column of the F_k vector is the F_{k+1} vector. The F_{k+1} vector represents the elements at the $(k + 1)^{th}$ level whose inputs are connected to the outputs of elements at the k^{th} level, and is found by using the following equation:

$$F_{k+1} = A * F_k$$

For the row assignment problem matrix calculations are also used. This involves the construction of a column adjacency matrix B which represents the connectivity between the k^{th} and $(k + 1)^{th}$ column. The elements of B are defined as:

$$b_{ij} = \sum \text{connection}(i, j) \begin{cases} \text{with module } i \text{ in column } k \\ \text{with module } j \text{ in column } k+1. \end{cases}$$

The objective is now to minimize the amount of signal crossings between adjacent columns by keeping the k^{th} column constant and changing the sequence of the $(k + 1)^{th}$ column. Since testing all permutations would involve $N!$ tests (see [MMA86]; N = number of modules present in the $(k + 1)^{th}$ column), a different approach is followed. The difference matrix, Z, is constructed from B and defined as:

$$z_{uv} = \sum_{j=1}^{q-1} \sum_{n=j+1}^q b_{uj} * b_{vn} - b_{vj} * b_{un}$$

If the columns of Z are sorted according to a decreasing number of negative signs per column, then that permutation of the modules corresponding with the new column sequence provides a reduction in the number of wire intersections. The new sequence of modules can be used for the next iteration, until no further improvements are found.

For routing, a simple channel routing algorithm is used. The horizontal (fanouts) and vertical

line segments needed to route the connections are grouped together, and non-overlapping segments are placed in the same vertical tracks. The optimization criterion is the number of tracks needed for routing.

Feedback and feed forward connections are preprocessed before normal routing can take place. Feedback loops are routed around the source module, and then treated as input by the channel routing algorithm. Feed forwards are replaced by feed-thru elements, which are treated by the router as single source single sink connections. Note that feed-thru elements correspond with the passthroughs used by the DAT-ASG.

The row sorting algorithm presented in this paper is based on the research presented in [MM83]. Its computational complexity is high, and given the time restraints for this assignment other methods are tried first. Another disadvantage of the row sorting algorithm is the fact that no special rules apply for passthrough elements. If the source element of a connection is placed at the logical grid position (i, j) , and the destination element at $(i, j+3)$, the two passthroughs needed for the connection might be placed in different rows than i . This is illustrated by figure 3.6. This would reduce the readability of the schematic.

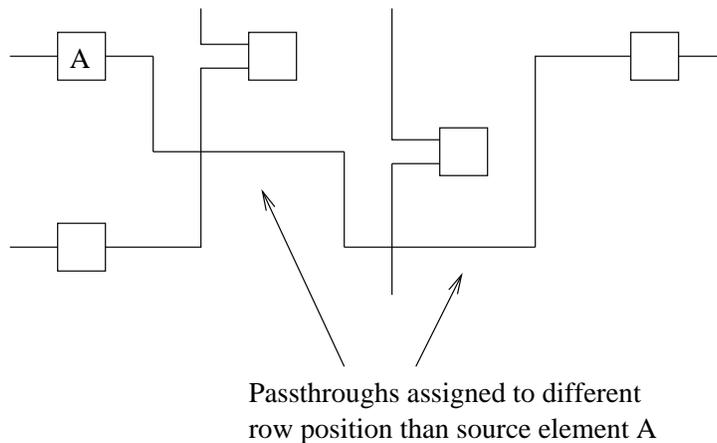


Figure 3.6: Example of the Decrease in Readability when Passthroughs do not have Fixed Logical Positions

3.7 "Automatic Generation of Digital System Schematic Diagrams"

See [AKM86] for the complete version of this paper.

This paper presents a complete and systematic solution to the ASG problem, using arbitrarily shaped modules and signal flow direction.

The paper starts by illustrating all factors and objectives in the ASG problem. It then explains that the ASG problem must be subdivided in order to be efficiently solved. The authors follow a two-step approach consisting of a logical phase and a geometrical phase. These phases

are identical to the logical placement and geometrical placement phase as explained in Chapter 2 which discusses the general solution to the ASG problem.

The logical phase is itself subdivided into several others: column and row assignment on a logical grid, module orientation (no fixed signal flow direction is assumed), and logical routing (assigning tracks to the connections and making sure connections do not overlap).

The geometrical phase also subdivided. It consists of final placement (placing multiple modules on the same logical grid space to compact the drawing area), final routing, terminal location (in case they may be arbitrarily placed around the module), and compaction (moving modules closer to each other).

A special feature mentioned in this paper is the ability to identify data buses. The connections forming the bus are drawn close to each other to form a data bus style routing.

The row assignment problem is solved by a cost minimization functions, using the amount of crossovers and bends in signal lines as factors.

This is one of the first papers in which the complete ASG problem is discussed, instead of a small part of it. The problem decomposition method suggested in this paper is now the accepted general solution method. These parameters must be set manually before starting the algorithm, depending on the user's personal preferences. However, the algorithms used to solve the subproblems are unfortunately not explained in details. If all possible combinations are tried and weighed, it would take $N!$ permutations for both placement and routing.

3.8 "GEMS: An Automatic Layout Tool for MIMOLA Schematics"

See [VV86] for the complete version of this paper.

This paper describes the Graphic Editor for MIMOLA Schematics (GEMS). The MIMOLA system is a collection of tools for digital design synthesis. One of GEMS' functions is to display a computer generated schematic, made from a network representation of the schematic.

The ASG problem in GEMS is divided into the placement and routing algorithms. In solving these problems, the same algorithms are used as those discussed in [LS89]. For more information see Section 3.10.

3.9 "VISION: VHDL Induced Schematic Imaging on Net-Lists"

See [RC87] for the complete version of this paper.

The VISION ASG system has been developed to increase the portability of VHDL (a hardware description language) designs. Though it is fairly easy to port a VHDL text description of a circuit from one system to another, it is much more complicated to port the

graphic representation of (parts of) the circuit used during design. VISION generates graphic representations at the logic gate level of a circuit described in VHDL.

VISION uses the net-list description generated from the VHDL description of a circuit. It first creates a connectivity matrix which stores all the connections that make up the circuit. This is a square matrix with equal row and column labels consisting of all the gate names present in the circuit. It then analyses the connectivity matrix to search for feedback loops within the circuit. If for example gates G1 and G2 form a latch, the connectivity matrix would show a connection from G1 to G2, and from G2 to G1. All feedback loops are removed from the schematic.

After removal of the feedback loops, each logic gate is assigned a column position based on their distance from the input pins (i.e. the leftmost position of the logic gate that preserves the left-to-right signal flow). VISION uses random initial row assignment.

After the initial gate placement, grid position calculations and pin assignments are made for each gate. Since from a functional point of view the pin ordering of logic gates is irrelevant, a maximum degree of freedom is present for pin assignment. Although the algorithm used for row assignment is not discussed, from studying the examples presented in the paper it appears to be the backward traversal algorithm which is discussed in [BN93].

As an additional rule, gates which have a large number of fanouts are placed at either the absolute top or bottom of the current column. The paper motivates this by stating that these signals usually contain clock or control information.

VISION also uses some local placement optimization rules to prevent unnecessary bends in signal lines. For example, if the source module G1 and destination module G2 of a connection have the same logical row coordinate, G1's output and G2's input terminal should have the same Y coordinate.

The final step involves routing. A standard channel width of five lines is used. The dominant routing optimization criterion is the number of bends in the signal lines. However since by now the grid positions have already been set I assume this only means that each connection is allowed to have a maximum of three line segments for routing. This is the minimum required to route a connections between two points with different x and y coordinates. The second criterion is the minimization of crossovers. The final criterion is the minimization of path lengths. Although there is no "best" optimization criterion, minimizing the amount of crossovers is the most commonly used method.

As is the case with the logical routing algorithm, no details of the actual channel routing algorithm used are described in the paper.

The connectivity matrix is an interesting concept to detect feedback loops. However, the feedback loops in DAT circuits have already been cut. The placement of gates with large fanout at the top or bottom of the schematic may be a nice option for the DAT implementation. The rule for adjusting the Y coordinate when a module's source and destination module have the same row coordinate is a useful optimization and is used by the DAT-ASG.

3.10 "From Network to Artwork"

See [LS89] for the complete version of this paper.

This paper describes a general purpose automatic schematic generator. Arbitrarily shaped modules are allowed, with terminals all around.

Used terminology in this paper:

- *Partition* - a set of modules connected to each other, forming a subset or partition of the schematic.
- *Strings* - a chain of modules connected to each other (the output of the first module in the string is connected to the input of the second, etc).
- *Box* - an area of the schematic with a few strings placed in it.

The ASG problem is divided into two problems: module placement and connection routing. Placement is done in several steps:

1. *Partitioning:*

The partitioning starts with selecting an appropriate seed module, usually the most heavily connected module. A partition is created by adding modules which are connected to the seed, to the partition. When the pre-defined partition size is reached, a new seed is selected from the modules which are not yet part of a partition. This process is repeated until every module is assigned to a partition.

2. *String Building:*

Second, each partition is further divided into strings. A string is a number of modules connected with each other in such a way that the left-to-right signal flow is preserved within the strings. If the end of a string within a partition is reached, a new string is started, using the modules which are not yet part of a string.

3. *String Box Filling:*

This step involves placing the strings in boxes. The area needed to draw the individual strings in the schematic is calculated in this phase.

4. *String Box Placement:*

In this step the string boxes forming a single partition are put together, and the total area needed to draw the whole partition in the schematic is calculated. The process of adding boxes to the partition starts with the box with the greatest number of modules. The next box added to the partition is the one with the greatest number of connections to the already placed box. For both boxes, a gravity center is calculated to find out the center of all connections inside a box. The second box is then placed in such a way the distance between both gravity centers is minimal, in the hope that this will minimize the total length of needed connections and the area needed for the complete partition.

5. *Partition Placement:*

The last step involves calculating the final position of each box in the schematic diagram. This is done in a similar manner as the placement of string boxes in a partition box.

The algorithm used for routing is a look-around algorithm. First, the whole area of the schematic is divided in logic wire segments in a similar manner as the logic grid used for logical routing. The algorithm then tries to make a straight connection between the source and destination terminal (using orthogonal lines). If an obstacle is encountered, the router tries the shortest route around the obstacle.

This is a general purpose solution for the ASG problem. It uses arbitrarily shaped modules with input and output terminals at all sides of the module, instead of the logic gate modules used in DAT. It is mostly focused on minimizing the area of the schematic, and the length of the connections used in the schematic. Also, it does not restrict placement and routing in such a way that the left-to-right signal flow is preserved. Further, it seems that a lot of extra data is needed to store information concerning boxes, strings, partitions and geometrical information. As a final comment, the routing mechanism described does not guarantee a solution if one exists. Earlier placement and routing decisions may have made it impossible to route certain connections, since there is no space left for it and there is no form of backtracking in the described algorithm.

3.11 "Structure Optimization in Logic Schematic Generation"

See [TL89] for the complete version of this paper.

The usual solution method for the ASG problem is to divide it into placement and routing. For placement, a logical grid is used. Every module is assigned a row and column position on this grid. Depending on these positions, the connections between the modules are routed.

This paper presents a method to find a near-optimal solution (a local optimum in the solution space) for the placement problem, and a method to find structural clusters within the schematic, such as branching trees, and adjust the schematic so that they are clearly visible for the human designer. It does not focus on the routing aspect of the ASG problem.

Due to the uniform size of logic gate modules, a grid system is used to abstract the placement and routing. The placement problem can be partitioned into a column and row assignment problem. Column positions are assigned such that the left-to-right signal flow is preserved. This means that feedback loops first have to be identified and removed.

The paper states that for the human designer the clarity of the schematic is improved if it contains symmetrical module clusters, symmetrical routing and a minimal number of cross-overs and bends. Since there can be no crossovers between aligned nets or well-formed symmetrical clusters, minimizing the number of crossovers is the primary optimization criterion for row assignment.

Since both placement and global routing (connections between modules in non-adjacent columns) determine the amount of crossover, the routing of global nets is modeled as a sequence of local nets (nets connecting modules in adjacent columns). These local nets are then connected

using inserted link nodes. The position of these link nodes in a column determines the position of the global net in this column. The row assignment problem is now reduced to a row sequence problem.

Row sequence determination focuses on the optimization of topology. It starts with a random sequence. Nodes are then swapped within their column to find a better sequence, using the method of simulated annealing to find a near-optimal sequence. If there are any crossing nets left after the optimization process, the connection with the greatest amount of crossovers is temporarily removed from the schematic. This process is repeated until there are no crossovers left.

The now planarised (without crossing lines) representation of the schematic is searched for topological structures. These structures include converging and extending trees. All nodes forming a structure are marked as such. The row position of each module is now recalculated. These positions are relative to the previous column. Once all relative module row positions have been calculated from left to right, the final positions of each module are calculated from right to left.

The authors have written an experimental program for the SUN 3 using the method described in this paper. As a typical performance example, a test circuit consisting of 37 gate nodes, 13 link nodes and 67 local nets takes 45 seconds to process.

This method might be of use for the DAT framework. The optimization method (simulated annealing) is usually slow and does not guarantee finding the optimal solution. It can however be one of many row assignment algorithms present in the final ASG implementation. The concept of structure analysis is interesting, but has to be implemented completely to be evaluated. Given the limited time available for this assignment this has not been done.

3.12 "Schematic Generation with an Expert System"

See [GS90] for the complete version of this paper.

This paper reports on a study of schematic generation using a Knowledge-Based System (KBS). To solve the ASG problem it has been divided into several subproblems, some of which are solved with conventional algorithms, and some of which are solved using knowledge rules.

The paper start by explaining the ASG problem, and showing that it is NP hard. This means that for large problems it is impossible to find the optimum solution, and only criteria can be specified. These criteria can be in the form of rules for a KBS.

To solve the ASG problem, is has been divided into several subproblems:

- *Horizontal Ordering Phase* - ordering of modules into columns
- *Vertical Ordering Phase* - ordering of modules within columns
- *Assignment of passthroughs*

- *Channel routing*

During the horizontal ordering phase, a column number is assigned to each module. Using a recursive graph traversal algorithm, the depth level of each module is calculated (see Section 2.4.1). For this algorithm to work, feedback loops must first be identified and removed. This is done by using a feedback loop detection algorithm, called Breadth First Search (BFS), which shows if feedback loops are present. If so, another algorithm called Depth First Search (DFS), is used to find the feedback arc. If complex feedback loops are present, more than one arc of the same loop may be used as the feedback loop. The KBS uses some rules to find which arc is best suited as a feedback arc (based on appearance rules). This feedback arc is then removed from the schematic and stored in memory. During routing, the final positions of each feedback loop are calculated.

Once a column number has been assigned to each module, the vertical ordering phase starts. This phase determines the sequence of modules within a column. It uses a barycentric algorithm, shifting the symbol to the gravity center (barycenter) or average row position of the nodes connected to its input. This algorithm can be extended to a three column optimization method, using not only the nodes connected to the input pins, but also those connected to the output pins.

Once all modules are assigned to a row and column position on the logical grid, passthroughs are located. Passthroughs are connections between modules in non-adjacent columns. In this paper passthroughs are modeled using "pseudo links", consisting of a single input and a single output module which performs no logical operation on the input. These pseudo links are all assigned to the same vertical position to increase the clarity of the drawing. A multi-column scan algorithm is used to find the optimal vertical coordinate for the passthrough. Once decided, the space used by the through is registered.

After passthrough assignment, the ASG problem is reduced to a channel routing problem. A set of rules in the KBS is used to find the best solution. These rules indicate when and where a fanout point may be placed, the minimal distance between signal lines, the minimal distance between signal line crossings, how many bends the signal line may have, how and where the feedback loops (which have been removed earlier) may be placed, etc. As these rules are not discussed in details, their use for algorithm design is limited.

The feedback loop detection algorithm (used to find out if the schematic contains feedback loops) is very fast and usefully, but in the DAT schematics feedback loops have already been cut and translated into internal inputs and outputs.

The barycentric column ordering algorithm seems very fast. If the results of this algorithm are not good enough, they can always be used as a starting position for an recursive optimization algorithm such as simulated annealing. A variation of it has been implemented in the DAT-ASG. Another variation, the bubble sort algorithm, has also been implemented in the DAT-ASG.

The passthrough assignment method used for this implementation seems useful, and a variation is used by the DAT-ASG. The difference is that the DAT-ASG not only assigns all passthroughs forming a single connection to the same logical row position, but as a second con-

dition assigns them to the same row position as their source module.

The channel routing method described in the paper is exclusively based on knowledge-rules, and is therefore of limited use for the DAT framework. It only states a number of optimization criteria. Some of these, such as the length of signal lines, are completely determined by placement. Others, such as the number and positions of crossovers, are of use but the rules governing them are not explained in depth.

3.13 "ASG: Automatic Schematic Generator"

See [YJ91] for the complete version of this paper.

In this paper a heuristic placement and routing method with a value propagation approach, the bubbling technique, is presented. Also, the left edge routing algorithm is modified with channel routing aspects to achieve compact channel size and regular channel routing. The primary considerations include circuit levelling, feedback loop detection, signal flow resolution, feedback and fanout signal routing, channel size estimation and track assignment.

In this paper, the placement and routing method is divided in two phases: logical placement and geometrical placement phase. The logical placement phase described in this paper includes both logical placement and logical routing and consists of the following steps:

1. Build a dynamic data structure representing all modules and connections.
2. Levelise circuit for column assignment and mark feedback loops.
3. Row assignment: sort each column (bubbling technique)
4. Detect fanout and feedback interconnections to make room for their routing in advance.
5. Do pin assignment for each module.

The geometrical placement (and routing) phase consists of:

1. Select the most heavily connected column level L and calculate its modules' temporary coordinates.
2. Include both columns adjacent to L , estimate the channel size between these columns and assign connections to tracks.
3. Repeat the previous step until all levels are included.
4. Use the temporary coordinates to do all final physical placement and routing.

Circuit levelization is done left to right by a levelizing algorithm. Each module has a variable called "inpins" indicating the number of input pins which have not been tested yet. It also has a variable "level" which is set to zero. Initially, this is set to the total number of input pins the element has. All modules connected to the input pins have a depth level of 2 assigned to them, and the inpins variable is reduced by one.

The algorithm then tests each module. If the "inpins" variable is zero it means that all input pins have been tested. The level of the module is now determined. All modules connected to its output pin are assigned its level plus one, unless they already have a higher level assigned to them. All modules connected to its output pin have their inpins number reduced by one. This is repeated until the inpins number of all modules is zero. For this to work, feedback loops have to be removed first.

After circuit levelization and feedback loop detection, each column is sorted using the bubbling technique. This is a value propagation algorithm. Each primary input is assigned a value based on its row position. From there on, each module is assigned the average value of the modules connected to its input pins. Once all modules are assigned values, each column is sorted in the order of these values. Only values that come from modules in the previous level are used for value propagation. An example of the bubbling technique for row assignment calculations is given in figure 3.7.

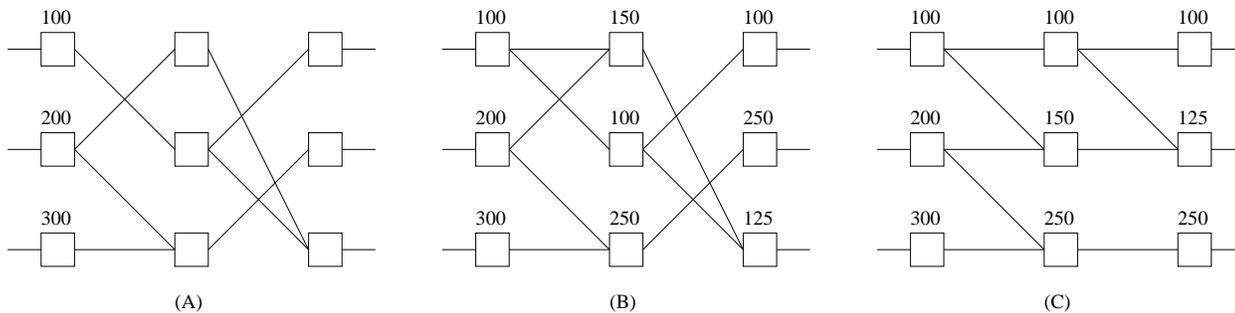


Figure 3.7: The Bubbling Row Assignment Algorithm. (a) Initial values, (b) after value propagation, (c) after row sorting.

During the next phase space is reserved for fanout trees and feedback loop routing. This is done by inserting pseudo modules in the schematic. These pseudo modules will make sure that enough empty space is present for the routing.

If the pin sequence of the modules is arbitrary, the optimal pin sequence (in terms of connection crossover reduction) of each module's input and output pins is calculated during the last step of logical placement.

In the geometrical phase the actual coordinates of each net and module are calculated. This starts with the column which is most heavily connected to the other columns. Once the positions of each module in this column have been calculated, the channel size (number of required tracks) at each side of this column is calculated. After this the positions of the modules in the two adjacent columns are assigned positions on the drawing which are relative to the modules in the previous column. All connections are assigned to a track and their relative positions on the drawing are calculated. This process is repeated in both directions until all columns have been processed. When all relative positions are known, the final positions of all modules and connections are calculated.

The channel estimation and track assignment algorithm is a modified version of the left edge algorithm used in VLSI channel routing. In this paper the channel is represented by two vertical lines (as is described in Section 2.4.1). Each routed connection consists of two horizontal line segments and a vertical segment. Initially, all vertical line segments are collected. These segments are sorted with respect to the position of the top of the vertical segment. Once sorted each net is assigned to its own channel. After initial assignments, connections with the lowest top edge are tested if they fit without overlap on the same track as nets with higher top edges. If so, both nets are placed in a single track. This is repeated until no more connections can be placed in a single track.

The most interesting item in this paper is the bubbling technique used for row assignment calculations. This method is useful and intuitive, and has been implemented in the DAT-ASG. The modified left edge channel routing algorithm described in this paper is similar to the algorithm currently used in the DAT-ASG. Although the method described here is slightly different (the DAT-ASG algorithm tries to merge connections from the start, instead of first assigning each connection to its own track and then merging them) the results are almost the same.

3.14 Conclusions of the Literature Reviews

When studying the algorithms for placement and routing described in the previous sections one notices that most papers follow a similar approach to divide the ASG problem into subproblems. For this reason, the general solution to the ASG problem has been described in the previous chapter (Chapter 2). This approach has also largely been used with the original DAT-ASG solution.

Some of the algorithms described in the literature which were already implemented in the original version of the DAT-ASG include:

- Barycentric Row Sort - The Fanin Centering algorithm present in the DAT-ASG is similar to the Barycentric Row Sorting algorithm. It calculates the average row position of the predecessors of elements and tries to assign elements to this position.
- The link nodes - Link nodes are similar to passthroughs and are already implemented in the DAT-ASG. The DAT-ASG has the additional condition that link nodes must be assigned to the same row position as their source elements.
- Left Edge Routing algorithm - The left edge routing algorithm for track assignment is also present in a slightly different form. The routing algorithm in the DAT-ASG is a first fit algorithm, which assigns connections to the first available track. The left edge routing algorithm also assigns connections to the first available track, but sorts them first according to the amount of vertical space they occupy in a track.

Useful new additions to the DAT-ASG algorithms include:

- Bubble Sort - A useful new algorithm from the literature which has been added to the ASG is the Bubble Sort algorithm. This is a modified version of the Barycentric Sort

algorithm. Barycentric Sort is similar to the fan centering algorithm described above: it tries to assign gates to a calculated position. Bubble Sort sorts these calculated positions and based on these positions, it determines the sequence of gates per column.

- **Circuit Input Pin Switching** - An algorithm to switch the sequence of the circuit's input pins has been added to the DAT-ASG. The existing implementation of this algorithm did not work, and it has been completely redesigned. This algorithm tries to assign gates to the row position of their successor(s).
- **Gate Input Pin Switching** - An algorithm to switch the sequence of a gate's input pins has been added to the DAT-ASG. By examining the row positions of a gate's predecessors, the gate's input pins can be sorted accordingly.
- **Module Swapping** - An algorithm has been added to swap pairs of modules when it improves the readability. The fanin centering and bubble sort algorithms try to reduce the number of signal line crossovers. In doing so, they sometimes reduce the readability of the schematic by increasing the number of bends. This algorithm increases the number of crossovers slightly while reducing the number of bends.
- **Several algorithms to highlight structures in the schematic:** the input cone, output cone, fanout free region, and a gate's neighborhood (region).

All algorithms which were added to the DAT-ASG are based on algorithms and ideas described in the reviewed literature. They have however been modified to fit into the DAT-ASG structure. Also, based on the reviewed literature, some existing algorithms have been improved for better results. For example, based on the knowledge of track assignment restrictions as offered by the VCG and HCG constraint graphs discussed in Section 3.2 the modified Left edge algorithm for track assignment has been fixed.

The implementation details of algorithms which were already present in the original DAT-ASG implementation are further described in Chapter 4. The shortcomings and errors in the original implementations are listed in Section 4.7. The newly added algorithms are described in Chapter 5.

The Original ASG Implementation

4

This chapter describes the original ASG implementation as designed by E.S. Sabbah (see [Sab94]) and improved by H. van der Linden. The purpose of this chapter is to describe how the original DAT-ASG functioned, since the current version is based on the original software implementation.

The chapter starts with Section 4.1 which showing how to start the ASG. Section 4.2 describes the ASG's input data, the circuit's netlist, and Section 4.3 describes the output of the ASG, the schematic diagram. The top level hierarchy of the source code is explained in Section 4.4. Section 4.5 introduces the ASG specific data structures. Section 4.6 gives an overview of the ASG algorithms implemented in the original ASG. Section 4.7 concludes this chapter with an overview of the used ASG solutions, a list of bugs and shortcomings in the original implementation, and a list of desired functionality which was missing in the original implementation.

4.1 Starting the ASG

The DAT (Delft Advanced Test platform) command line environment has a number of ASG related commands. An overview of these will be given in appendix A. The easiest way to draw the schematic of a circuit is to first load it into DAT using the 'circread' command. You can then either save it as a file using the 'draw' command, or display it on the screen using the 'xdraw' command.

The 'circread' command orders DAT to read the circuit, which may be specified in a number of different formats. The circuit will then be stored into memory using DAT's internal circuit representation. This is also the input data for the schematic generator.

The 'draw' and 'xdraw' commands call the ASG routines which calculate the position on

the drawing surface of every element and connection. These positions are stored in memory, and then used to either save the schematic picture to a file in postscript format or to draw the schematic of the circuit on the computer monitor.

4.2 The ASG Input Data

Once the circuit is read into DAT's memory, it is stored as a graph using two basic classes, namely:

- Class element
- Class connection

Elements are stored in memory in element class objects. These objects contain member functions and data fields to store their type and connectivity information. Elements include logic gates such as AND, OR, INVERT, as well as virtual elements such as circuit inputs and outputs, and fanout points. Each element (except circuit inputs) has one or more inputs: one for fanout points and circuit outputs, and one or more for logic gates. They (except circuit outputs) also have one or more outputs: one for logic gates and circuit inputs, and two or more for fanout points. Pointers to the connections attached to the element's individual inputs and outputs can be obtained using member functions. From the ASG's point of view, only four different types of elements exist:

- Circuit inputs
- Circuit outputs
- Fanout points
- Logic gates (the rest)

Throughout this report, these four basics will be displayed in schematics using the symbols in figure 4.1. Fanout points, however, will only be drawn when they are discussed or when they are absolutely needed to prevent misunderstandings concerning connections.

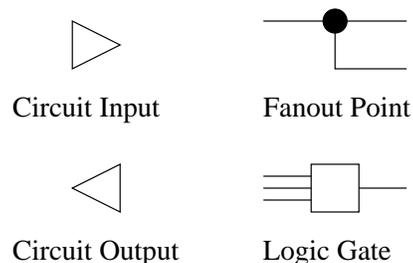


Figure 4.1: Schematic Symbols used in this Manual.

Connections are stored in memory in connection class objects. They have member functions that determine the elements they come from and lead to.

With these two classes, the entire circuit graph can be represented. This forms the input data for the ASG, which calculates element and connection positions based on the circuit's connectivity information.

4.3 The Graphical Output

The ASG's graphic output contain a number of different possibilities. By invoking the 'xdraw' command from DAT's command line, the circuit currently contained in memory will be displayed on the screen in a window. Figure 4.2 gives an example of what you will see on your screen when displaying a schematic. The functions of the buttons at the top of the window are:

- *Quit* - close the window and exit DAT.
- *Save* - save the displayed schematic to a postscript file, including the current zoom factor.
- *Back* - return to DAT's command line without closing the schematic's display window.
- *Stop* - return to DAT's command line and close the schematic's display window.
- *ScaleUp* - increase the zoom factor of the displayed schematic.
- *ScaleDown* - decrease the zoom factor of the displayed schematic.
- *DispOnce* - toggle DISPLAYONCE setting on/off. When this setting in 'on', DAT immediately returns to the command line after drawing the schematic.

Two scrollbars are also present to display schematics larger than the current display windows.

4.4 ASG Source Code Hierarchy

The source code for the ASG functions, like the rest of the DAT framework, is written in C++. This section gives an overview of the different files containing the ASG source code:

- The ASG specific objects and their member functions are declared in 'xdraw.dcolumn.cxx' and its header file 'xdraw.dcolumn.h'.
- All ASG specific calculations for the exact positions of all elements and connections are described in the file 'circdraw.cxx' and its header file 'circdraw.h'.
- All ASG specific graphic functions needed to display the schematic on the computer monitor are described in the file 'xcircdraw.cxx' and its header file 'xcircdraw.h'.
- All functions for saving the schematic as a postscript file are described in the file 'pscircdraw.cxx' and its header 'pscircdraw.h'.

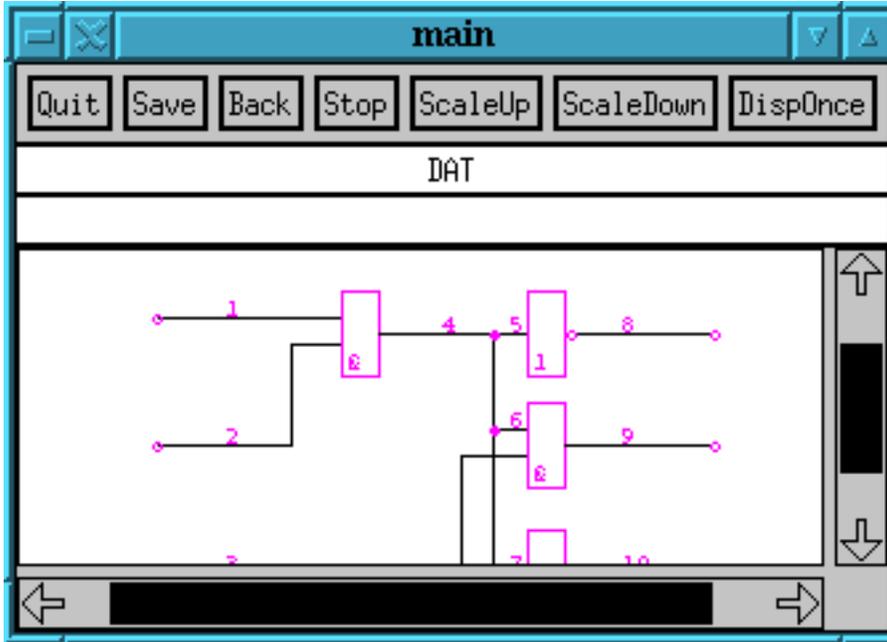


Figure 4.2: Screen-dump of the ASG's graphic output.

4.5 ASG Specific Data Structures

The ASG uses a number of custom data objects to store logical and geometrical placement and routing information of the elements and connections. These objects are defined in the file 'circdraw.dcolumn.cxx' and its header 'circdraw.dcolumn.h'. This section gives an overview of the custom ASG data objects which are used in the DAT-ASG implementation.

Note that these two files also describe a number of functions which were either canceled or never used in the original DAT-ASG, which was used as the basis for further development. Since they were mostly undocumented in the source code, and unmentioned in the manual, their meaning and intension remains largely unknown. They have been left in for possible future use.

DAT's ASG contains a number of assumptions which simplified the design. One of these assumptions is that each channel contains the same number of tracks. A channel is the space between two columns of elements where connections are drawn (routing). By assuming a fixed width for the channel and the elements, the X coordinate calculations of individual elements were simplified. This assumption causes a direct linear relation between the logical grid's column position of an element, and the actual X coordinate on the drawing surface:

$$X - coord_i = GridColumn_i * (ChannelWidth + ElementWidth)$$

The disadvantage is that all channels take up the amount of space needed for the channel with the most tracks.

A second assumption is a fixed vertical size for each position on the logical grid, whether it

is for a single connection skipping the current column (a passthrough, see fig 2.1) or for a 5-input logic gate. This eliminates the need for complicated calculations of the Y coordinate of individual elements. It is enough to know the logical row position to calculate the actual Y coordinate on the drawing surface:

$$Y - coord_i = GridRow_i * ElementHeight$$

These two simplifications also eliminated the need to check whether elements would overlap with other elements and passthroughs, since they all have their own individual space. The disadvantage of this is that elements, and especially passthroughs, take up more space than is absolutely needed, making the schematic larger. This is illustrated in fig 4.3.

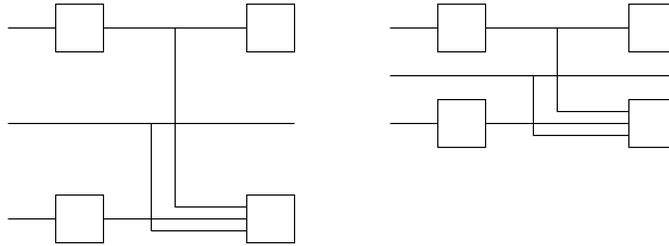


Figure 4.3: Drawing of the same Schematic with and without fixed Element Area's.

The logical column position of each element in the schematic is already known (it is the depth level of each element, or in other words the maximum distance in logical gates between the element and the circuit's inputs). All ASG specific information about elements, as well as reserved spaces for passthroughs, is stored in an array 'columns[]' of 'dcolumn' objects. Each entry in the array represents logical placements information about one column of the schematic. The class 'dcolumn' is a daughter of the classes 'dblock_coll' and 'passthru_coll', and represents a single column of elements and passthroughs in the schematic 'dblock_coll' is a collection of elements, with member functions to manipulate the logical order of elements within the column. 'passthru_coll' is a collection of passthroughs, which are used as virtual elements for connections between elements in non-consecutive columns.

Once the logical position of each element in the row is known, the exact coordinates can be calculated since there is a direct linear relation between the logical grid position and the actual coordinated on the schematic diagram. These coordinates are stored in the arrays 'elem_X' and 'elem_Y'.

Figure 4.4 gives an overview of the objects which are used to store the logical and geometrical placement information. The arrows between the boxes represent inheritance between the different object classes.

The logical and geometrical routing information of all connections is stored in the array 'channels', consisting of 'channel' type objects. The class 'channel' is derived from class 'track_coll' which in turn is a collection of 'track' class objects. The object 'track' represents a single track in a channel. It contains a collection of 'track_seg' structures, which store the connection numbers and the Y coordinates of the edges of the vertical space they occupy within the track, limiting the amount of data that needs to be stored during track assignment to three

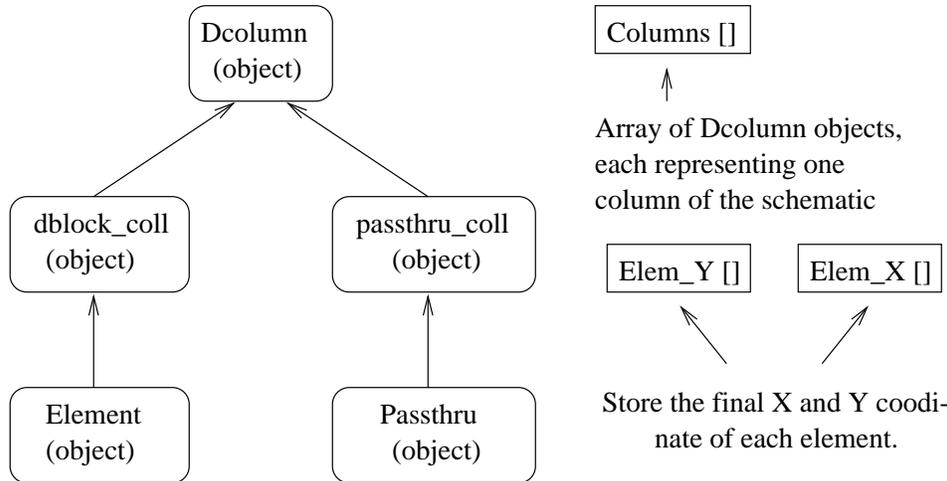


Figure 4.4: Data structures used for Placement Information.

integers per connection. The disadvantage of this is that the Y coordinates of the edges of a connection's vertical line segment must already be known during track allocation. But since each position on the logical grid takes up the same space, the logical row positions of the source and destination elements of a connection are enough to calculate the exact Y coordinates of these edges.

The final geometrical routing information is stored in the array 'con_SEGS'. For each connection in the circuit it contains a pointer to an object of type 'segment_coll', which is a collection of 'segment' class objects. The 'segment' class is a single line with starting and ending coordinates. On a schematic, each connection consists of one or more line segments.

Figure 4.5 gives an overview of the objects which are used to store the logical and geometrical routing information. The arrows between the boxes represent inheritance between the different object classes.

In order to improve readability of the schematic, each fanout point is marked on the schematic by a dot. The positions of these dots are stored in an object of type 'coord_coll'. This is a collection of 'coordinate' structures. Each 'coordinate' stores the exact position of a dot.

4.6 ASG Functions and Algorithms

From an ASG designer's point of view, the real work is done in the 'calc_coords' function in the file 'circdraw.cxx'. This function calls the individual steps for calculating the exact coordinates of all the elements and connections that make up the schematic diagram. The following subsections will describe the various algorithms which are used in the 'calc_coords' function to solve the ASG problem.

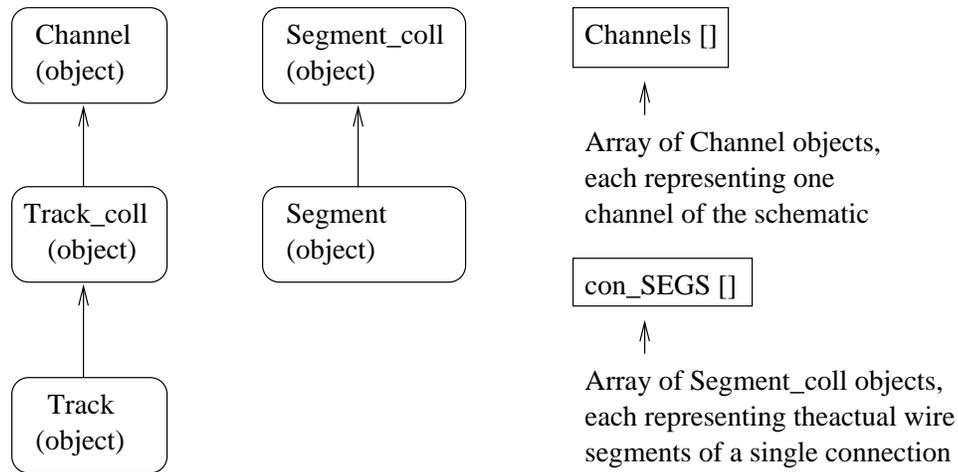


Figure 4.5: Data structures used for Routing Information.

4.6.1 Logical Placement Algorithms

The first step in the ASG solution is to assign each element to a position on a logical grid i.e. assigning a column and row coordinate to each element.

After the circuit has been read by DAT, each logic gate is assigned a depth level. This depth level is the maximum number of logic gates between the current gate and the inputs, and it is used as the column position on the logical grid.

For the initial placement on the logical grid a recursive algorithm called 'backward traversal' (see [BN93]) is used. The backward traversal algorithm is illustrated in figure 4.6. For each output of the circuit, the connection attached to it is traced back to its source logic gate. This gate is then assigned to the first available row position in the column indicated by the gate's depth level. After this, its input pins are traced back to their source gates, which in turn are assigned to the first available row position in their columns. This is repeated until all connections are traced back to the circuit's inputs.

After initial row assignment, a few adjustments are made. The first adjustment to the initial placement is called Horizontal Alignment, and it is illustrated in figure 4.7. Starting from the first column (directly to the right of the circuit inputs), it adjusts the row position of blocks such that they move to the same height as, or lower than, their highest predecessors. When a block is pushed down, the blocks under it are pushed down with it. As an example, if an element's predecessors have a logical row position of 2 and 3, the element's row position will be adjusted so that it is 2 or greater.

After horizontal alignment, a process called Fanin Centering takes place, which is illustrated in figure 4.8. For each element, the average row position of its predecessors is calculated. This is done by examining the minimal and maximal row position of elements in the previous column which connect to the element under investigation, and then calculating the middle of this interval. If the new position is greater than the current position, it can be moved if there is free space at that position.

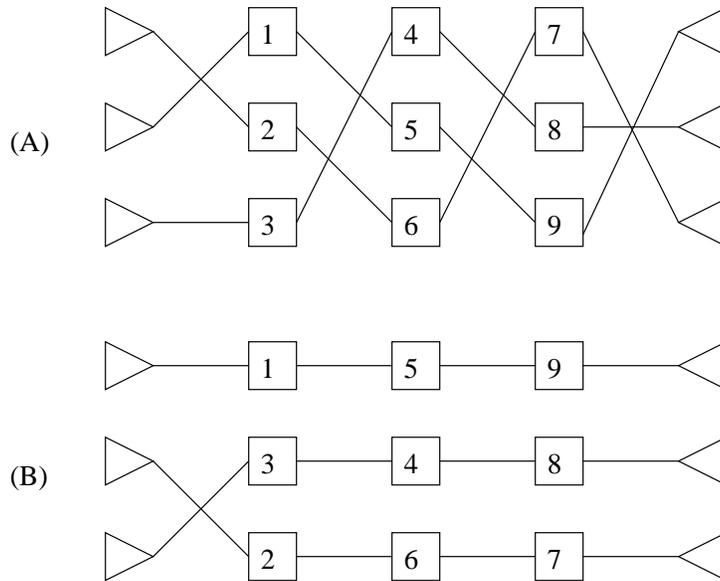


Figure 4.6: The Backward Traversal Algorithm: (a) before and (b) after applying.

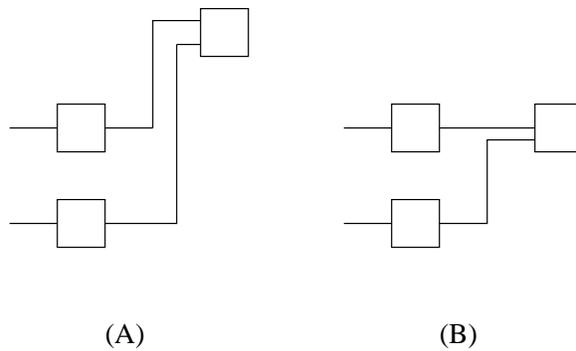


Figure 4.7: The Horizontal Alignment Algorithm, (a) before and (b) after applying.

The next step in logical placement is adjusting the row position of the circuit inputs such that they become equal with the highest element they feed. This process is called PI vertical order adjustment, and illustrated in figure 4.9

After row position adjustments, room is made for passthroughs. The process of passthrough space creation is illustrated in figure 4.10. Passthroughs are connections between elements in non-consecutive columns. By making room for them in the logical placement phase, it is guaranteed that there will be room later when the connections are routed. The room for the passthrough is made at exactly the same row position as the source element of the connection that needs a passthrough. Note that this can undo earlier made improvements: if a passthrough is created at the position which was calculated as the fanin center position of a gate, this gate and all gates under it are pushed down, undoing the fanin centering algorithm partly. This is solved

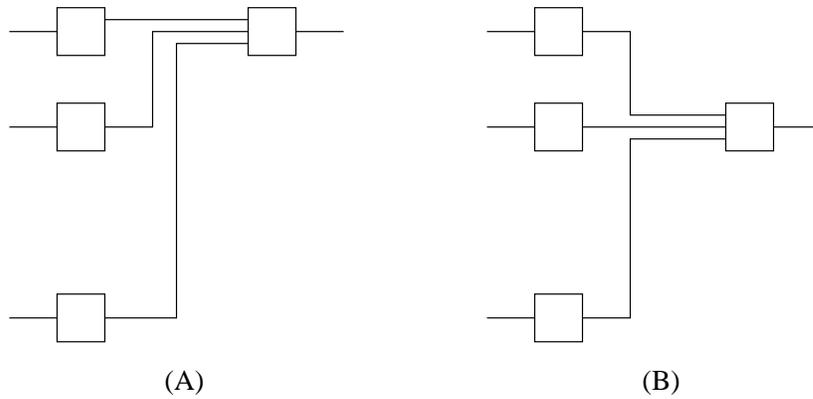


Figure 4.8: The Fanin Centering Algorithm, (a) before and (b) after applying.

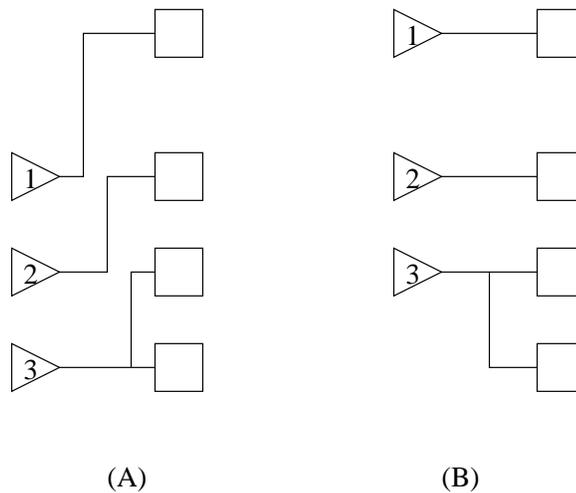


Figure 4.9: The Inputs Vertical Adjustment Algorithm, (a) before and (b) after applying.

partly by executing the Fanin Centering again after room has been created for the passthroughs.

After making room for the passthroughs, the row positions of the circuit outputs are recalculated such that they become equal to the row position of their predecessors. This process is called PO vertical order adjustment, and is somewhat similar to the PI vertical order adjustment.

Note that after the logical placement phase has been completed, all elements, including fanout points and passthroughs, have a column and row position assigned to them. Fanout points have the same row and column position as the source element of their stem.

4.6.2 Logical Routing Algorithms

According to the general solution method for the ASG problem as presented in Chapter 2, in this phase the connections are assigned to tracks one channel at a time. However, DAT's

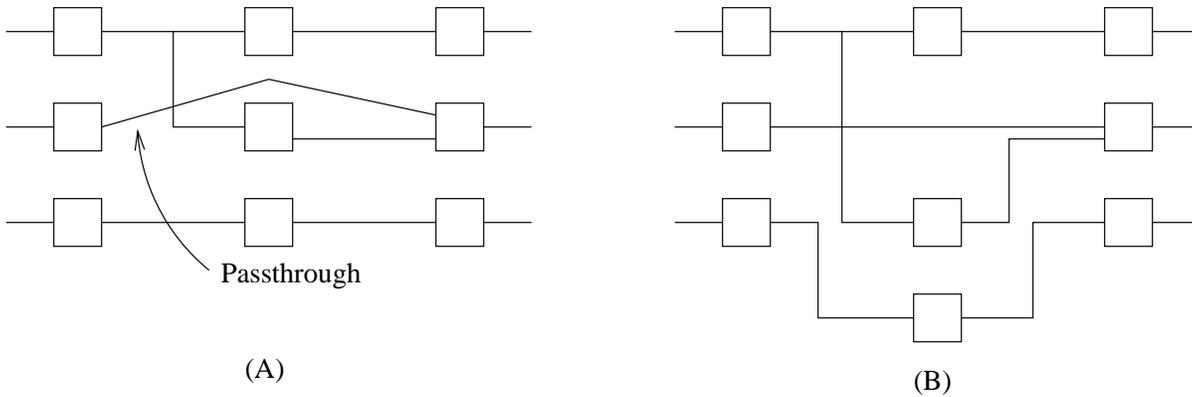


Figure 4.10: Making room for Passthroughs, (a) before and (b) after applying.

ASG uses a different strategy to route connections. First of all it assumes a fixed size area for each position on the logical grid. After logical placement it immediately starts calculating the actual coordinates of each element on the schematic (skipping the logical routing phase as described in Chapter 2). The DAT-ASG then bases its routing on these coordinates, and assigns connections to tracks one at a time. The routing algorithm looks at the connection's source and destination element. It calculates which channels are crossed between the connection's source and destination element and handles routing for those channels

Although the routing is based on module coordinates, it is still a two step process. The first step is assigning track numbers to connections by reversing the vertical space they need based on the coordinates of their source and destination modules. The second step calculates the actual line segments on the drawing which every connection needs for its routing.

The track assignment used by the DAT-ASG uses the final element coordinates on the drawing surface of the schematic for track assignment. Since this chapter discussed the original ASG implementation, which starts calculating geometrical placement immediately after logical placement, we will now move on to that. The complete implementation of the routing problem will be explained in detail in Section 4.6.4.

4.6.3 Geometrical Placement Algorithms

Once all logic gates have a position assigned to them on the logic grid, the actual coordinates on the schematic diagram can be calculated. For this, the original ASG implementation uses a straightforward process. Since one of the assumptions is that each position on the logical grid corresponds with a fixed area on the drawing surface of the schematic diagram, the actual coordinates are calculated by multiplying the logical column and row position with the column and row width in pixels.

A similar process takes place for the circuit's inputs and outputs (PIs and POs). First, their positions are calculated as if they are normal logic gates. After this, their Y coordinates are adjusted so that they are positioned in the middle of the block in the next/previous row.

For fanout points (GFANs) a slightly different process takes place. During logical placement, they are assigned to the same column as the source element of their stem. During geographical placement, they are assigned to the same Y coordinate as their source element's output pin. Their X coordinate is made equal to that of the first track of the channel between the stem's source element and the next one (see Section 4.6.4 for stem and branch track assignment).

Although passthroughs can be considered as elements without any logic function or delay, they can also be considered as part of the routing problem. DAT's ASG takes the same approach to passthroughs as it does to fanouts: they are elements without physical dimensions. The passthrough gets the coordinates of the middle of the area on the drawing surface which corresponds with their logical grid position.

As a last adjustment step, each element's inpins are checked. If an inpin's connection comes from a element with the same logical row position, the block's Y coordinate is adjusted such that the inpin is drawn at the exact same height as the other element's output pin. This prevents two bends in the signal line. The use of this step is demonstrated by figure 4.11.

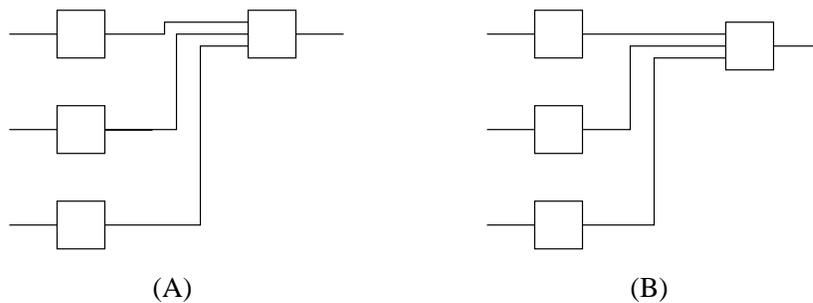


Figure 4.11: Fine-tuning a Block's Y Coordinate, (a) before and (b) after applying.

4.6.4 Geometrical Routing Algorithms

Once the exact coordinates of all elements on the schematic's drawing surface are known (the circuit inputs and outputs, the logic gates, the fanout points and the passthroughs) the actual exact positions of the line segments forming the connections are calculated.

First, the track assignments of all channels are calculated. For each connection, the channels it crosses are calculated. Per channel the Y coordinate of the elements (whatever their type) in the previous and next column are calculated. For connections spanning multiple columns they will usually be the same be the same (for a passthrough section, or connection without any vertical line segment). These Y coordinates are then used to find a track where the vertical space needed (between the two Y coordinates) would not overlap with another connection's vertical space that is already assigned to this track. Note that this effectively is a first fit algorithm. If there is space, a track segment is added to the track, storing the connection number and the vertical edges.

Branch connections are routed differently. Instead of using the connection number of the individual branches, the stem's connection number is used for the track assignment of the

stem and all its branches. This causes multiple branches to elements in the same column to use the same track number: the total vertical space needed to route all branches of a stem to a single column is calculated, and one track request is done for the total vertical space required.

Figure 4.12 shows how the track assignment works. Connection 1 is a stem, which branches into connection 2 and 3. The track space required consists of the total vertical space needed to route all branches of connection 1 in this single channel. The required track space is listed as occupied by connection 1, and is assigned to track 1. Connection 4's source and destination pin have the same Y coordinate, so no track assignment is required since it neither needs space nor does it overlap with another connection. Connection 5 can also be assigned to track 1 since the vertical space it requires does not overlap with the connections already assigned to track 1. Connection 6 is assigned to track 2 since by now there is no space for it in track 1. Connection 7 is assigned to track 3 since by now no space is left in track 1 and 2.

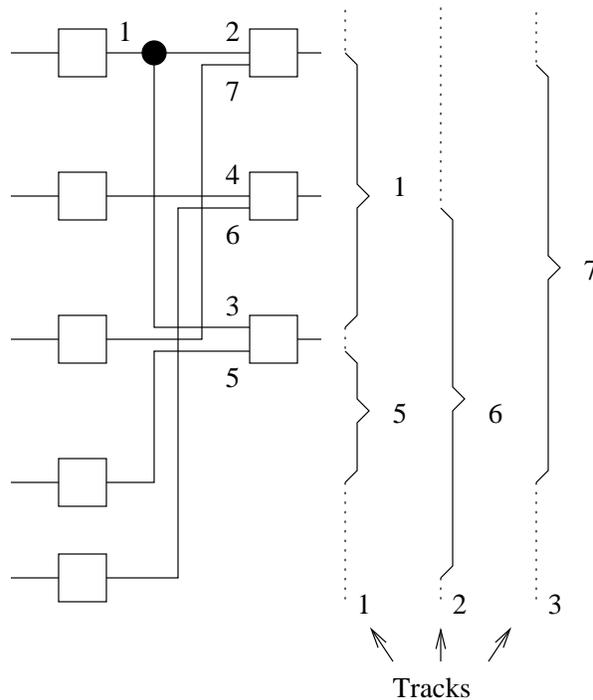


Figure 4.12: An Example of the Track Assignment Algorithm.

The second step is calculating the exact coordinates of the vertical and horizontal line segments that form the connection. This is again done by looking at the exact coordinates of a connection's source and destination point. The three different line segment routing cases are:

1. *Straight line connection* - This is the simplest form of routing. When the source and destination Y coordinate of a connection are equal, the connection is a single horizontal line segment. This type of routing is used for normal connections between two module in consecutive columns when the source element's output pin and the destination element's input pin have the same Y coordinate. It is also used for passthrough and stem type connections.

2. *Single Vertical Segment* - If a connection has different source and destination Y coordinates, the track it is assigned to is looked up. With the ASG implementation, the number of tracks per channel is fixed (whether they are used or not). The routing consists of two horizontal and one vertical line segments. First, a horizontal line segment from the source element to the assigned track. Second, a vertical line segment through the track. Third, a horizontal line segment from the assigned track to the destination element's input pin. The X coordinate of the track is calculated using the multiplication of the track number and the track size, plus an offset from the last column of elements. The Y coordinates for the vertical segment are those of the source element's output pin and destination element's input pin.
3. *Branch Connections* - Fanout branches are routed as normal connections, using the fanout point as the source element. This has the effect of line segments of different connections on top of each other, something which is not a problem as long as individual connections are not highlighted (using colors). This issue will be discussed later in 5.2.4.

4.6.5 Fanout Point Algorithms

In order to improve readability of the schematic, fanout points are marked by drawing a dot on top of them. This prevents mistaking a signal crossing for a fanout point. The dots are calculated by an algorithm that check each channel track by track, track segment by track segment. For each track segment, the connection number is checked to see if it leads to a fanout point (since branches are stored by their stem's connection number). If so, the fanout point's branches are all checked to see if dots need to be placed in this track.

Two basic fanout point categories can be differentiated:

1. The fanout point is positioned at the same height as the output pin of the stem's source element. This is the case for fanout point 1 in figure 4.13. If a stem has branches to elements in columns i and j , and $j > i$, then a fanout point is needed in column i .
2. The fanout point is positioned on one of the stem's vertical line segments. This is the case for fanout point 2 in figure 4.13. The Y coordinate of output pin of the stem's source module is $Y1$. If a stem branches to multiple element in the same column whose input pins have Y coordinates $Y2$ and $Y3$, a fanout point is drawn at $Y2$ if:

$$(Y3 > Y2 > Y1) \vee (Y3 < Y2 < Y1)$$

In both cases the X coordinate of the fanout point equals the X coordinate of the track to which the stem connection is assigned in this channel.

4.7 Conclusions about the Original ASG Implementation

This section first gives an overview of the algorithms used to solve the ASG problem. It then presents a list of bugs and shortcomings of the original ASG algorithms. It ends with a list of functionality which was missing in the original implementation.

Algorithms used to solve the ASG problem:

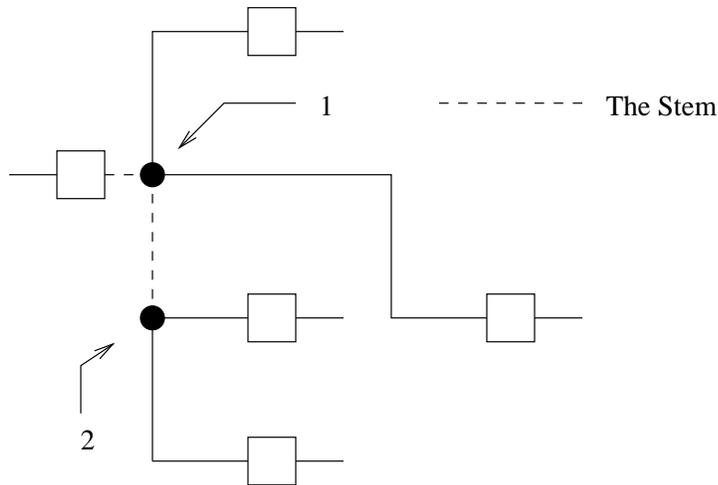


Figure 4.13: Example of Fanout Point Categories.

1. Calculate the position on the logical grid of each module in the circuit. Use the level of each module as their column position. Calculate the row position of all modules using the 'backward traversal' algorithm.
2. Adjust the row positions of all blocks per column, using the following algorithms in sequence:
 - (a) Horizontal Alignment - set the row position of each module equal or below its highest predecessor.
 - (b) Fanin Centering - if possible set the row position of each module equal to the average row position of its predecessors.
 - (c) PI Adjustment - if possible set the row position of the circuit's inputs equal to that of its successor(s).
3. Make room for the passthroughs:
 - (a) Push down all blocks in the column to make space for the passthrough directly behind its source element. Do the same for all columns that need a passthrough for this connection.
 - (b) When done with this column, call a Fanin Centering algorithm to recalculate the row position of all non-passthrough modules.
4. Recalculate the row positions of all modules using Fanin Centering.
5. PO adjustment - if possible set the row position of the circuit's outputs equal to the row position of its predecessor(s).
6. Calculate geometrical module positions using a direct mapping from logical column/row positions to geometrical x/y coordinates.

7. Adjust Y coordinates of blocks for straight line connections.
8. Assign connections to tracks.
9. Geometrical Routing - For each connection calculate the line segments forming a single connection.
10. Calculate positions of fanout points.

While examining the existing DAT-ASG, a number of bugs and shortcomings were detected. They are:

1. The Fanin Centering algorithm used to recalculate the row positions of non-passthrough modules calculates the wrong row position. It does not calculate the average row position based on the sum of the row positions of the module's predecessors, divided by their number. Instead it calculates $((SUM + N/2)/N)$, which gives wrong results. This manifests itself by calculating a higher row position than it should, which means the modules are drawn at a lower position in their column than they should. Figure 5.1(a) shows the old situation. When calculating the row position of the block in column 2 it should result in placement in row 1. By placing it in row 2, all modules connected to its output will also be placed one row lower than they should. This creates a 'downward drift' of modules through the schematic.
2. The Fanin Algorithm used to recalculate the row positions of non-passthrough modules uses a very simple placement strategy. When it has calculated a module's new row position based on its predecessors' row positions, it checks if space is available at that position. If there is no space available, nothing is done. It would be better if the algorithm considers the whole interval between the original position and the newly calculated position, since any position in this interval will improve upon the existing situation.
3. The PI adjustment algorithm does not do anything. This algorithm should try to place the circuit's inputs at the same height as the elements they feed. If they lead to a fanout point they should be placed as close as possible to the vertical position of any of the branch destinations.
4. The PO adjustment algorithm tries to place the circuit's inputs directly behind their predecessors. If however their predecessor is a fanout point, all branches will be placed on top of each other. This was patched later, but the patch forgot to make the old position available again for PO placement. This causes unnecessary empty positions in the circuit output column.
5. The track assignment algorithm assigns connections to tracks by checking only if they do not have vertical overlap. Vertical overlap happens when vertical line segments of different connections (partly) overlap each other. It does not check if connections have horizontal overlap (overlapping horizontal line segments). Horizontal overlap causes confusion since it is no longer clear which line segment is part of which connection.

6. The geometrical routing algorithm uses the same algorithm to route every connection from its source to its destination module. When routing fanout stems it calculates the routing from the source module to the fanout point (virtual module). It routes the fanout branches from the fanout point to the destination modules. As a consequence, branch connections partly overlap each other. This becomes visible only when individual branches are given different colors.

A second consequence is the 'ghost line' problem. The ghost line problem is illustrated in figure 4.14(A). It is not immediately clear which line segments are part of the stem and which line segments are part of branches. In this example a module in column 1 leads to a fanout point with branches leading to modules in columns 2, 3 and 4. The thicker line is how the stem connection appears when highlighted. The small circle behind the module in the leftmost column represents the virtual fanout module as used by the original DAT-ASG. In 4.14(b) the correct line segment forming the stem connection is drawn as a thicker line.

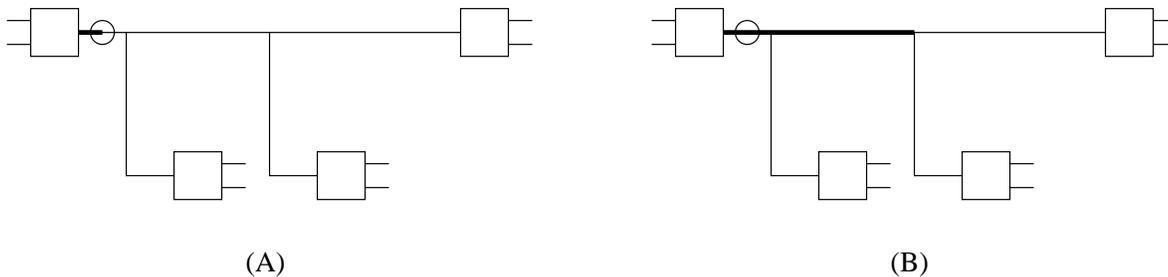


Figure 4.14: Example of the Ghost Line problem, (a) the original situation and (b) the correct situation

7. Various graphic issues - the original DAT-ASG had a number of graphic errors and limitations:
 - (a) On initialization of the channel data structure each channel has ten tracks. When these tracks remain unused, space is still reserved for them in the schematic. When more than ten tracks are needed the ASG runs out of space to route the connections.
 - (b) If a gate has more than three inputs, the input connections are drawn so close together that it becomes hard to see which goes where, unless the zoom factor is increased dramatically.
 - (c) When drawing connections starting at the circuit's input pins, there is a small gap in the first line segment.
 - (d) The first part of any line segment starting at an element's output pin is drawn in the element's color instead of the connection's color.

The existing DAT-ASG also lacked functionality which would have improved the program:

1. The initial placement of the modules is the single most important step in the DAT-ASG process. From there, all other calculations take place. A second method for initial placement would generate a completely different schematic. Based on the reviewed literature the Bubble Sort method (see [YJ91]) was selected as a second initial placement algorithm.

2. Add an algorithm to detect pairs of gates whose row positions should be switched. This should be done for pairs of gates whose predecessor elements have each other's logical row position.
3. Since algorithms such as horizontal alignment improve the schematic in some cases, and worsen it in others, the user should be able to switch all algorithms on and off.
4. For test algorithm development it is useful to highlight structures in the schematic. Four structure were selected for inclusion in the DAT-ASG:
 - (a) Input Cones - all connections encountered when the specified connection is tracked backward through its predecessor elements, stopping when a gate is encountered which is more then N levels away from the start element.
 - (b) Output Cone - all connections encountered when the specified connection is tracked forward through its successor elements, stopping when a gate is encountered which is more then N levels away from the start element.
 - (c) Fanout Free Region - all connections encountered when the specified connection is tracked backward through its predecessor element, stopping only when a fanout point is encountered.
 - (d) Region - all connections encountered when the specified connection is tracked forward and backward through its predecessor and successor elements, stopping only when a gate is encountered which is more then N levels away from the start element.

Improvements of the existing ASG Algorithms

5

The original DAT-ASG contained a number of bugs and shortcoming. They are listed in section 4.7, along with a list of lacking functions. This chapter gives an overview of all to the changes to the original DAT-ASG. Section 5.1 lists the changes made to the data structure. Section 5.2 and its subsections show the changes made to the existing ASG algorithms. Section 5.3 describes the ASG algorithms which were added to the DAT-ASG. Section 5.4 describes the graphic related changes. Section 5.5 lists the commands and the switches which were added to the DAT-ASG command line environment.

5.1 Changes to the Data Structure

The original ASG implementation was used as the base for the current one, and large parts of the original program have been left intact. As a consequence, the data structure has remained mostly the same.

The following functions have been added to the data structure of the DAT-ASG:

- `add_with_pos(...)` has been added to class `track`.
- `add_track_with_pos(...)` has been added to class `track_coll`.

Changes have been made to the following functions:

- `is_addable(...)` of class `track`.
- `calc_x_coord(...)` of class `track_col`.

These changes to the data structure are explained below in detail.

Examining the original track assignment algorithm showed it was not capable of detecting horizontally overlapping connections. The overlap problem is explained in detail in section 5.2.2. To solve the overlap problem, connections had to be un-assigned from their channels and reassigned to a new track where they do not cause any new overlaps. When a connection is reassigned to a channel, the program searches for a track the connection can be assigned to. A track segment, storing the vertical interval needed for this connection, is added to this track.

This process of un-assigning and reassigning connections leaves empty positions in the track segment list (the track) and the track collection list (the channel). Functions dealing with these lists expect every position in the lists to be filled with data. If an empty position is encountered, the program halts. This is solved by shifting track segments and tracks to lower positions in their lists. For this purpose two additional functions were added to the data structure:

1. The class 'track' has been extended with the function 'add_with_pos', which allows the addition of track segments at a specified position in the track segment list.
2. The class 'track_coll' has been extended with the function 'add_track_with_pos', which allows the addition of tracks at a specified position in the track list (the channel).

A small adjustment has been made to the member function 'is_addable' of the 'track' class. This function checks if a track segment (the vertical space needed to route a connection in a channel) can be added to a track. Connections that do not take up any vertical space are also registered as occupying track space, with their begin and end vertical position being equal. Connections without a vertical segment can always be added to a track, since they never overlap with other connections. The original implementation overlooked this, and performed overlap checks for these cases as well.

The member function 'calc_x_coord' of the track_coll class (a track_coll is a channel) is used to calculate the x coordinate of all tracks in a channel. The original implementation assumed tracks were numbered from right to left. There was no documentation motivating this choice, as the more logical choice would be a left to right numbering, which is also used for column numbering. The function has been changed, and track numbering is now done from left to right.

5.2 Improvements to existing ASG Algorithms

The original ASG implementation contained a number of bugs and shortcomings, which are listed in Section 4.7. The following sub-sections discuss all changes of the existing ASG algorithms which were made to correct these bugs and shortcomings.

5.2.1 Improvements to Logical Placement Algorithms

While studying the original DAT-ASG implementation, the following bugs and shortcomings were noticed in the logical placement algorithms:

1. *Fanin Centering algorithm* - the fanin centering algorithm tries to place elements at the average logical row position of their predecessor elements. The calculation of the average logical row position was incorrect, causing the elements to be placed lower than they should. Also, the algorithm tried to place elements at the calculated position only, instead of considering alternatives as well.
2. *PO Order Adjustment algorithm* - the PO adjustment algorithm, which calculated the optimal logical position of the circuit's outputs, originally placed the circuit outputs directly behind their predecessor element. However, when the predecessor is a fanout point, all circuit outputs connected to the fanout point's branches will be drawn on top of each other. This was patched later on by checking if the position is available. This patch introduced a new problem: if the optimal position of a circuit output pin is blocked by another output, it could not be moved to this position. This has been solved by combining the original algorithm with the patched version.

The remainder of this section discusses these problems in detail.

Fanin Centering Algorithm:

During logical placement, space is reserved for connections between elements in non-consecutive columns. These connections are called passthroughs. After reserving row positions for passthroughs, an element centering algorithm is executed. This algorithm, called Fanin Centering, tries to improve the row position of logic gate elements based on the row positions of their predecessors. The calculations are done in the function 'do_centering'. This function in turn executes the function 'get_center_position' to calculate the optimal row position of an element based on the average of its predecessors' row positions.

The original implementation of the fanin centering algorithm calculated the row position using the sum of all predecessors' row positions, using the following equation:

$$\text{unsigned int ROW} = ((\text{SUM} + (\text{N}/2)) / \text{N})$$

In this formula, N is the number of inputs of the element. However, when the average position of the predecessors' row position lies in the middle of two row positions, the calculated position should be the rounded down result. For example, if an element has predecessors with row positions 1 and 2, this formula would calculate ROW=2, as shown in figure 5.1. The reason for preferring the lower row position number 1 to the higher row position number 2 is that row positions are numbered from top to bottom (as in matrices). By assigning the element to the lower row position, the schematic does not drift off downwards, which would create extra empty space in the schematic diagram. The algorithm is now a straightforward average position calculation (ROW=SUM/N).

The function 'do_centering' itself has also been adjusted. Originally, it tried to fit elements at the calculated center position, by checking if that position is empty and no passthrough has been reserved there. The algorithm now checks the complete interval between the calculated center position and the original position, starting at the center position. This reduces the amount of empty space on the logical grid, as shown in figure 5.2.

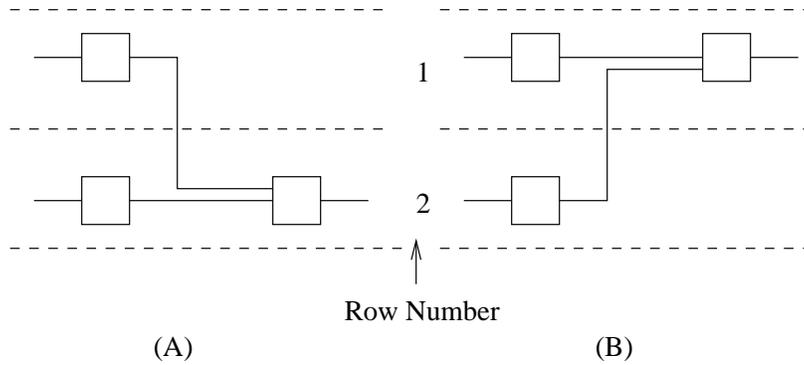


Figure 5.1: Row Calculation during Fanin Centering, (a) before and (b) after Improvement.

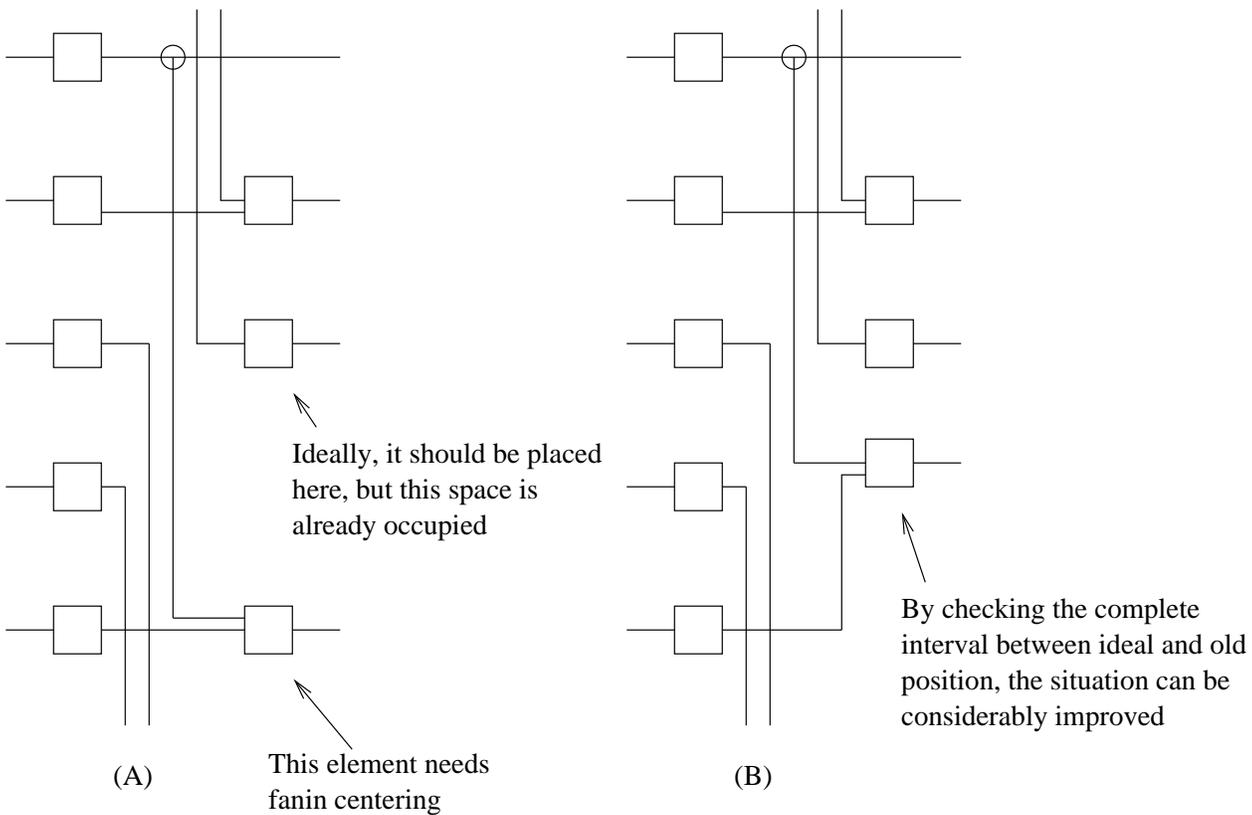


Figure 5.2: Slot Assignment Possibilities, (a) before and (b) after Improvement.

PO Order Adjustment Algorithm

The final step during the logical placement phase of the ASG involves adjusting the vertical position of the circuit's output pins. This is done by the function 'adjust_POs_vert_order'. The original implementation consisted on placing each output pin at the same vertical position of its predecessor element. However, in the ASG database, fanout points are also represented as elements (GFAN's). This meant that if a signal splits into branches, all branches are drawn on

top of each other.

This problem has been solved using a two step solution. First, all output pins whose predecessor elements do not consist of fanout points are placed at the same vertical position as its predecessor. Second, all output pins whose predecessor elements consist of fanout points are placed. This is done by first testing if the same vertical position the fanout points occupies is available in the output pin column. If it is already occupied, the output pin column is examined by linear search to find the empty row position closest to the one taken by the fanout point.

5.2.2 Improvements to Logical Routing Algorithms

During logical routing connections are assigned to tracks. The original DAT-ASG track assignment algorithm overlooked the problem of horizontal overlap. When two connections have overlapping horizontal line segments, as shown in figure 5.3(a), it becomes unclear which line segments belong to which connection. This problem has been solved in the current implementation. The details of this are described in this section.

The implementation of the DAT-ASG assumes that each position on the logical grid is of equal geometrical size on the schematic. This means that each element and passthrough occupies a fixed size area, and that each position on the logical grid can be linearly mapped to a geographical position on the schematic drawing. As a consequence of this, the original ASG has no real logical routing, but instead uses the geometrical coordinates of the individual elements to route connections. However, connections are still assigned to tracks before actual routing takes place.

The track assignment algorithm used by the ASG assumes that connections can be assigned to the first available track in the channel where they need to be routed. The track assignment algorithm only checks if a connection's vertical line segment will overlap with other connections assigned to the same track. For this it uses a simple track assignment algorithm which looks at each connection in the circuit, finds out which channels it uses, and then assigns it to the first available track in those channels where it doesn't overlap with the connections already assigned to this track.

However, apart from overlap within a single track, connections can also overlap with other connections that will be assigned to later tracks. If a connection A ends at the vertical position where connection B starts, connection B must be assigned to an earlier track or the connections will overlap each other on the schematic, making it impossible to track connections over the schematic. This is demonstrated by figure 5.3.

This problem has been solved by adding functions to the ASG code which correct the overlap. Each channel is tested from left to right, by taking a track segment (the vertical space occupied needed to route a single connection) and testing if it will overlap with track segments of track to the right of it. If so, both the track segment under investigation (A) and the track segment with which it overlaps (B) are removed from the channel. The segment causing the overlap (B) is then assigned to the first position where it causes no new overlap with later track segments. Track segment (B) is first assigned to a new track since it needs to be assigned to a lower track number than track segment (A), as shown by figure 5.3. After assigning track segment (B) to a track, track segment (A) is assigned to the first available track after (B) where it causes no new

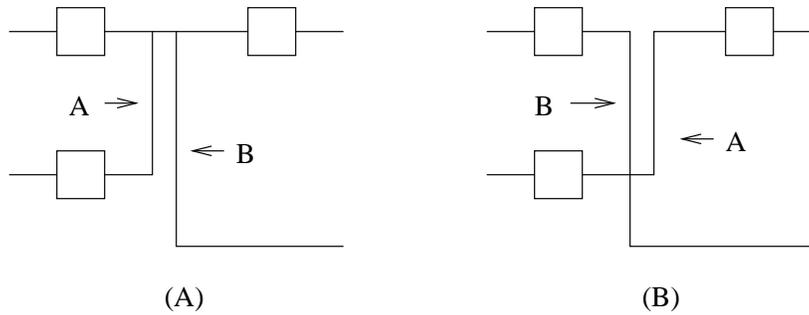


Figure 5.3: Assigning Connections to Tracks, (a) original and (b) corrected Implementation.

overlap.

This cut and paste process can leave empty slots in the track list and the track segment list. This happens when all segments in a track are removed, leaving an empty track, or when a single track segment in a track has been removed and there are other segments left with higher track segment numbers, leaving an empty track segment. All ASG algorithms dealing with tracks assume that all tracks contain at least one track segment, and that track segment lists contain no empty positions. The reason is that the functions examining track and track segment lists use a variable, indicating the number of items in the list, to determine valid list entries. If for example a list contains 3 items, stored in positions 1, 2 and 4, the program will crash when examining position 3. This problem has been solved by shifting all track after the empty track one position to the left (tracks are numbered from left to right). The same is done with the track segment list.

5.2.3 Improvements to Geometrical Placement Algorithms

The original geometrical placement code has mostly been left unchanged. During the geometrical placement phase an element's column and row position on the logical grid are transformed into an element's actual X and Y coordinates on the drawing surface. Geometrical placement in the ASG is a straightforward mapping between the logical grid position and the geometrical position. Based on the logical grid position an element occupies, the coordinates are calculated. This was done to speed up the design process for the original implementation. Except for increasing the amount of empty space on the circuit's schematic, and thereby increasing its size, it has no disadvantages. One improvement which has been made to the geometrical placement code is decreasing the default channel size from 10 tracks to 5. For small schematics this decreases the total area considerably, and when more tracks are needed for routing the channel size is increased by the ASG.

5.2.4 Improvements to Geometrical Routing Algorithms

During geometrical routing the line segments needed to route each connection on the schematic's drawing surface are calculated. While studying the existing DAT-ASG geometrical routing algorithms, the following bugs and shortcomings were noticed:

1. *Overlapping Branch Connections* - the original DAT-ASG routed all branches of a fanout point from a fixed point on the schematic's drawing surface: all branches started their

line segments from this point. This meant that line segments of different branches were overlapping each other. When all branches have the same color this cannot be noticed. But when different branches are given different colors, this can be noticed clearly.

2. *Ghost Line problem* - when routing a fanout point's stem and branches, it is unclear where the line segments of the stem end, and the line segments of the branches begin.

These two problems are closely related, and have been solved by completely rewriting the routing code for stem and branch type connections.

When connections are routed they are then stored per connection as a collection of line segments. This is done by looping over all the connections in the circuit, calculating their source and destination point, and then calculating the required line segments. The original ASG assumed that all connections are drawn in the same color. However, the current version allows the user to give distinct colors to individual connections, allowing the user to highlight parts of the schematic. This demonstrated a shortcoming in the original ASG geometrical routing code.

A fanout point is in many aspects treated as an element without physical size, and the fanout point is the source element of its branches. By using the same code for the geometrical routing of fanout stems and branches as is used for other connections, the stem is only drawn to the virtual fanout point, and the branches are partly drawn on top of each other. As long as both stem and branches have the same color this cannot be seen. This is demonstrated by figure 5.4. All branches coming from the fanout point are routed using either one or three line segments. Line segments of different branches partly overlap each other, and depending on which one was last given a different color, the displayed results will vary. The problem is that it is not clear which line segments belong to the stem and which to the branches. This problem is referred to as the 'ghost line' problem.

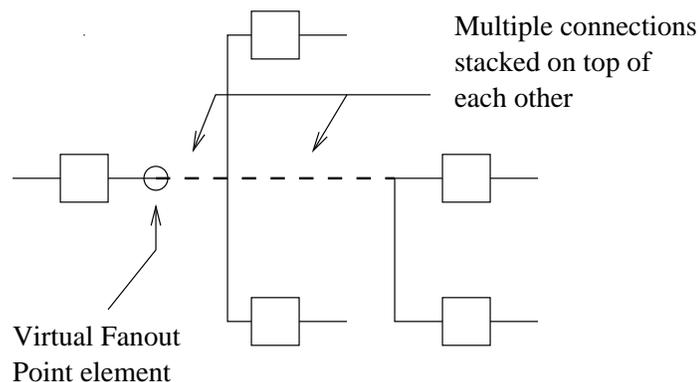


Figure 5.4: Example of multiple Branches from a single Fanout Point Stacked on Top of Each other.

To solve the ghost line problem, the geometrical routing code for stem and branch connections had to be completely rewritten. This has been done by adding new functions to the ASG. If a connection is a stem or a branch, geometrical routing is performed by the new routing functions. The rest of the connections are routed by the original geometrical routing algorithm.

The first step in routing the stem and branch connections is to calculate the horizontal line segment of the stem. Two cases can be distinguished:

1. Of all the fanout point's branches, the branch destination column furthest away from the fanout point is not the only branch destination in its column. The stem's horizontal line segment stops in that column at the track position the stem has been assigned to (branches are not assigned track positions. Instead, the stem itself is assigned track positions for all branches.) See figure 5.5 (a).
2. Of all the fanout point's branches, the branch destination column furthest away is the only branch destination in its column. The stem's horizontal line segment then stops in the 2nd furthest column of the branch destination columns. See figure 5.5 (b).

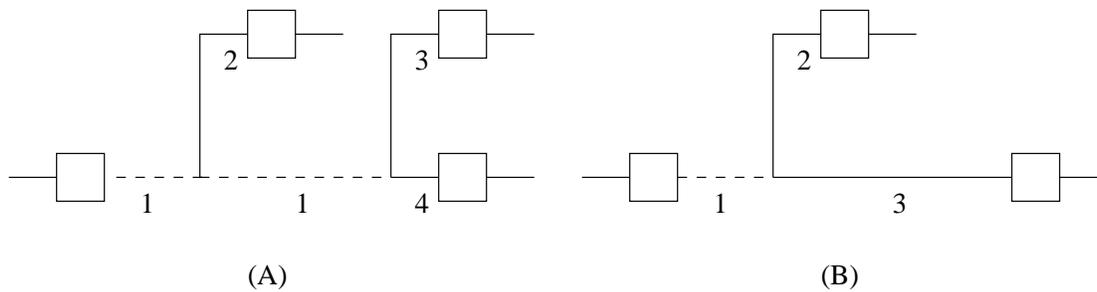


Figure 5.5: The Stem's horizontal Line Segment.

Routing the stem connection's horizontal line segment is solved by examining which branch leads to the element furthest away of the stem's source element. Once this connection is known, the branches are examined again to find which one leads furthest away from the stem's source element, but this time excluding the branch found earlier. This channel to the left of this branch's destination element will be called the reference channel. The X coordinate of the track to which the stem is assigned in this channel will be used for the end point of the stem's horizontal line segment.

Once the stem's horizontal line segment is calculated, all branches of the fanout point are examined. Three major categories of branches can be differentiated:

- If a branch is the only one that leads to a certain column, the line segments needed to route it to the stem are all part of the branch connection, as demonstrated by connections 2 and 3 in figure 5.5 (b). The branches are routed using either one (connection 3) or two (connection 2) line segments.
- If there are multiple branches leading to elements in the same column, and a branch leads to the only element above or the only element below the stem, all segments needed to route it to the stem's horizontal line are part of the branch connection. This case is shown in figure 5.6 (a). Line segments which are part of the stem are drawn as dashed lines.

- If there are multiple branches leading to elements in the same column, and there are multiple branch destinations located at the same side of the stem, the routing depends on the branch destination's position. If for example the branch leads to an element below the stem's horizontal line two cases exist:
 - If the branch leads to an element below the stem's horizontal line, but there are other branches of the stem leading to elements in the same column which have even lower positions, the branch is routed to the track which is assigned to the stem. From there a vertical line segment is routed to the stem's horizontal line, which is part of the stem's line segments. This is illustrated by connection 1 in figure 5.6(B). The same applies to multiple branch destinations above the stem's horizontal line, as illustrated by connection 1 in figure 5.6(C).
 - If the branch leads to an element below the stem's horizontal line, which has the lowest position of all branch destinations in that column, the branch is routed back to the track assigned to the stem. From this point a vertical line segment, which ends at the vertical position of the second lowest branch destination, is also added to the branch's line segments. This is illustrated by connection 2 in figure 5.6(B).

The same applies for multiple branch destinations in the same column which are above the stem line, as is illustrated in figure 5.6(C).

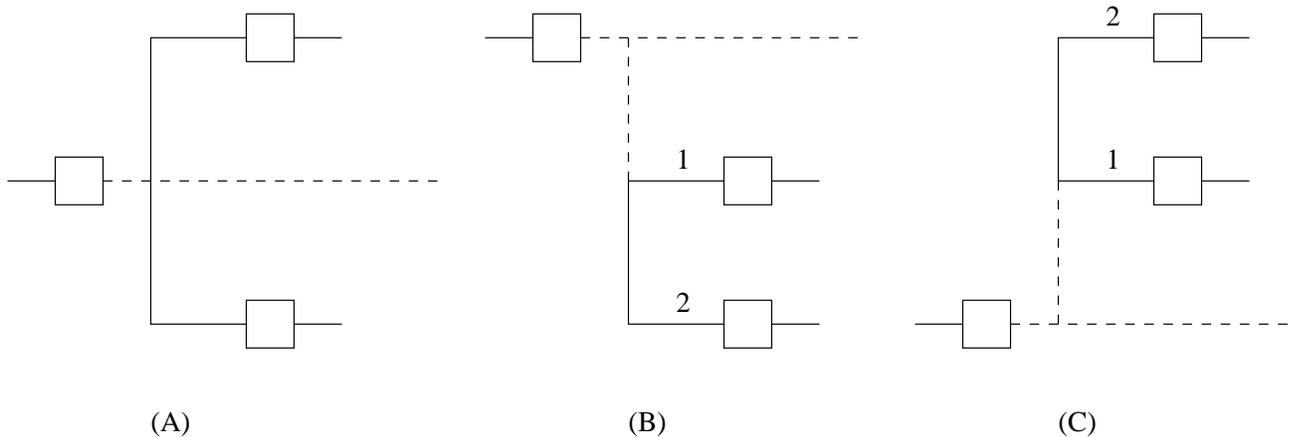


Figure 5.6: Examples of the geometrical Routing of Branches.

5.3 Additional ASG Algorithms

This sections discusses two algorithms which have been added to the DAT-ASG:

1. *Bubble Sort algorithm* - the bubble sort algorithm has been added to the DAT-ASG as an additional algorithm for initial placement. The final appearance of the schematic is to a large degree dependent on the initial placement algorithm: this is the only algorithm which does not base its results on the existing ordering of the circuit's elements.

2. *Pairwise Swap algorithm* - the pairwise swap algorithm has been added to switch the positions of pairs of elements when this improves the readability of the schematic. It checks for a reduction in the number of bends in the connections leading to the elements, at the cost of a small increase in the number of signal line crossovers.

The remainder of this section discusses the details of these algorithms and their implementation.

Bubble Sort Algorithm:

The original ASG implementation used the 'backwards traversal' algorithm (see Section 4.6.1 for more details) for the initial logical placement. The results of this algorithm are highly dependent on the pin assignment sequence of the elements in the schematic. The backwards traversal algorithm travels through the circuit starting at the circuit's outputs, and then recursively following the input pin connections of each element it visits for the first time. When an element is visited for the first time, it is assigned the first available row position in the schematic column corresponding with its depth level.

This means that the sequence in which elements are connected to a gate's input pins determines to a large amount the placement of the gates. Therefore, the backwards traversal algorithm is mostly suited as an initial placement method, after which other algorithm are used to fine tune the placement.

Studying the ASG related literature showed the bubble sort algorithm (see Section 3.13 for details) as the most promising placement algorithm. Therefore, this algorithm has been implemented as an alternative for the backward traversal algorithm. The user of the DAT-ASG can choose which initial placement method is used through a settable.

The original bubble sort algorithm has been adjusted for the DAT-ASG. The paper in which the algorithm is published specifically states that gates only propagate their values if their output connection leads to a gate in the next (adjacent) column. If it skips one or more columns (passthroughs) the value is not propagated. The paper only mentions this briefly, and does not explain what should be done in this case.

Since the DAT-ASG assumes that passthroughs are placed directly behind their source element, the row position of the source element can be used by the bubble sort algorithm. By calculating the bubble values one column at a time, starting from the most left column, it is guaranteed that all elements attached to the input pins have already been assigned row position on the logical grid. The bubble values of each element is then calculated using the row position RP of elements connected to its input pins, using the following equation (N = number of input pins of the element):

$$BubbleValue = \frac{\sum_{i=1}^N RP_i * 100}{N}$$

This is also illustrated by figure 5.7.

The final algorithm has been implemented as follows:

1. Assign input pins to column 1 in the same sequence as in the circuit description file.
2. Calculate the total number of elements per column.

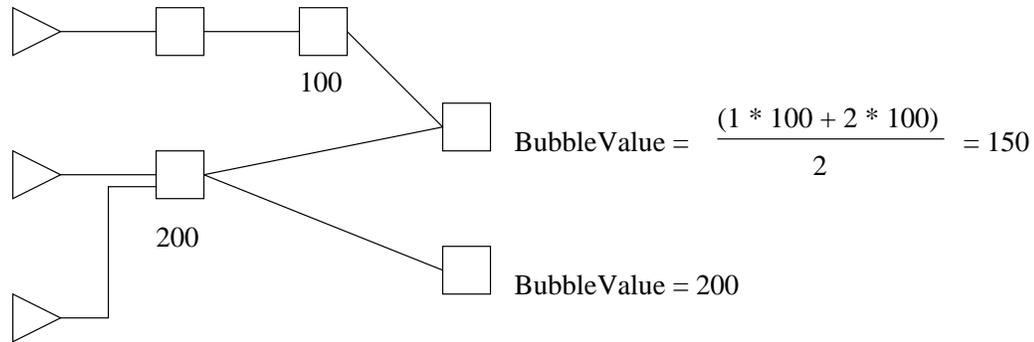


Figure 5.7: Example of the Adjusted Bubbles Sort Algorithm.

3. Calculate the row sequence per column from left to right, starting with the column after the input pins, doing the following:
 - (a) Create an array to hold the BubbleValue of each element in the column.
 - (b) Calculate the BubbleValue of each element in the column, using the row position of the elements connected to the element's input pins.
 - (c) Sort the BubbleValues in ascending order.
 - (d) Using the sorted Bubble Values, assign the corresponding elements to the column in the same sequence.

Pairwise Swap Algorithm:

A second addition to the ASG algorithms is the pairwise swap algorithm. The purpose of this algorithm is swap the positions of two elements A and B, if one of the elements connected to A's input pins is positioned in the previous column and has the same logical row position as element B, and vice versa. This is shown in figure 5.8. Note that although the number of line intersections has increased (from 1 to 2), the overall readability has improved due to the reduction of bends in signal lines (from 6 to 2).

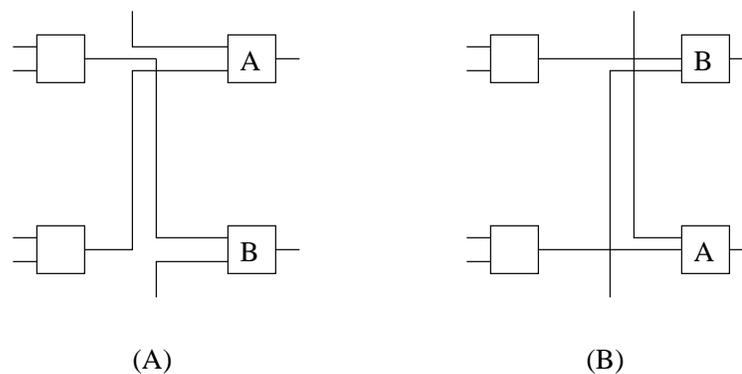


Figure 5.8: Example of the pairwise swap algorithm, (a) before and (b) after applying.

The pairwise swap algorithm is executed after space has been reserved for the passthroughs, so that the elements are already at their final positions. It has been implemented as follows:

Function 'The elements should be swapped' (elements A,B) :

for (each input of element A)

for (each input of element B)

SourceA = the element connected to A's input

SourceB = the element connected to B's input

if ((SourceA in the previous column) AND

(SourceB in the previous column) AND

(row position (SourceA) = row position (B))) AND

(row position (SourceB) = row position (A))

return (true)

Function 'Pairwise Swap':

for (each column in the circuit)

for (each possible pair of elements in the column)

if (the elements should be swapped) do (swap the element pair)

The third addition to the DAT-ASG is the option to swap the circuit's input pins.

NOTE 1:

The following algorithms have been added to the DAT-ASG, but have not yet been documented in this section:

1. Algorithm to draw the circuit's input pins at the optimal position based on the row positions of their destination modules.
2. Algorithm to draw the modules' input pins in the optimal sequence base on the row position of their predecessors.

NOTE 2:

Additional algorithms which will probably be added to the DAT-ASG in the coming days:

- An algorithm to place modules leading to fanout points with a high number of branches at the top of their column.
- An algorithm to locally exchange routing line segments if it reduces the amount of crossover.

5.4 Additions to the Graphic Output

In order to highlight structures in the schematic, the DAT-ASG allows users to give different colors to connections. To extend this functionality, four different types of structures can be highlighted using a single order. These structure are:

- Input Cone: all connections up to N levels away, which are attached to an element's input pins, are highlighted.

- Output Cone: all connections up to N levels away, which are attached to an element's output pins, are highlighted.
- FFR (Fanout Free Region) Cone: all connections which form the Fanout Free Region of an element are highlighted.
- Region 'Cone': all connections up to J levels backwards and K levels forwards, which are attached to an element's input and output pins, are highlighted.

Figure 5.9 shows the four different type of cones. The connections which are part of the cone are indicated by dashed lines. Figure 5.9(a) shows a one level deep input cone. Starting from the element with output connection 'j' and back tracking its input pins, all connections leading from this element to elements which are one depth level/column away are drawn. This excludes connection 'i', which leads to an element which is more then one level away.

Figure 5.9(b) shows a one level deep output cone. Starting from the element with output connection 'j' and forward tracking its output pin, all connections leading from this element to element which are one depth level/column away are drawn.

Figure 5.9(c) shows the fanout free region (FFR) of connection 'j'. The connection is traced back to its source element, and from there all input pins are traced backwards and added to the FFR until either a circuit input or a fanout point is reached.

Figure 5.9(C) shows the one deep region of connection 'j'. It combines both in input cone and the output cone of connection 'j'.

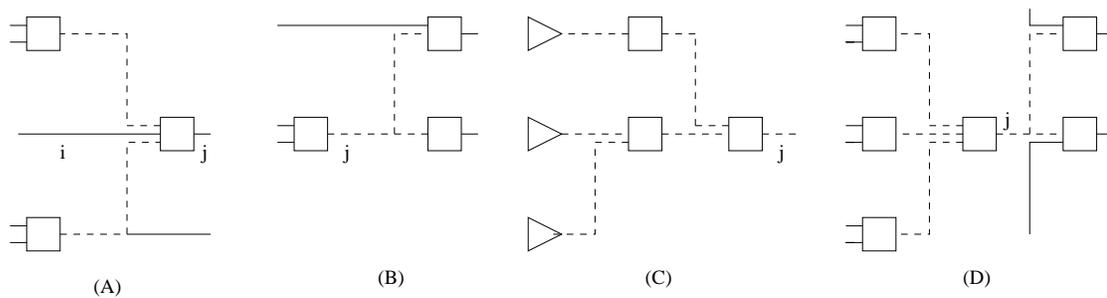


Figure 5.9: The four Cone types. (A) Input cone, (B) Output cone, (C) FFR cone, (D) Region 'cone'.

Studying the cones shows that the region cone is a combination of an input and an output cone. It also shows that the FFR cone is a special case of an input cone: instead of stopping the highlighting N levels away from the start element, the algorithm stops only when a fanout point or circuit input is encountered. This means that an algorithm for calculating an input cone, with a switch to stop at FFR's, and an algorithm for calculating an output cone are enough to implement all four cones.

The output cone algorithm first calculates at which column it should stop, and then recursively follows the output connection. For each element it encounters, it checks if its column position is still in range. If so, the connection leading to it is highlighted, and each output of the element it leads to is processed in a similar manner. If a fanout point is encountered, each branch is checked in a similar manner, and the stem is highlighted.

The input/FFR cone is calculated in a similar way. For each input pin, the incone algorithm checks if the column position of the element connected to it still in range, or in the case of an FFR it checks if the input pin is not connected to a fanout point. If so, the connection is highlighted, and each input of the element it leads to is processed in a similar manner.

5.5 Improvements to the Command Line Environment

In order to display the cones, which were described in Section 5.4, the following five commands were added to DAT's Command Line Interface:

- `xdrawinputcone`
usage: `xdrawinputcone < connr > < levels > [colornr]`

Set the drawing color of an input cone which is `< levels >` levels deep and start at connection `< connr >`.

- `xdrawoutputcone`
usage: `xdrawoutputcone < connr > < levels > [colornr]`

Set the drawing color of an output cone which is `< levels >` levels deep and start at connection `< connr >`.

- `xdrawffr`
usage: `xdrawffr < connr > < levels > [colornr]`

Set the drawing color of an Fanout Free Region which starts at connection `< connr >`.

- `xdrawregion`
usage: `xdrawregion < connr > < nback > < nforward > [colornr]`

Set the drawing color of a region cone which goes `< nback >` levels back and `< nforward >` levels forward and start at connection `< connr >`.

- `xdrawresetconncolors`
usage: `xdrawresetconncolors [colornr]`

Set the drawing color of each connection to the default color (0) or to `[colornr]`.

All these commands allow the user to set a color number. If no color number is specified, the default color is used. The results of the coloring operations can be viewed after invoking the `xdraw` command again.

The second improvement to the Command Line Interface is the addition of a number of settable scalars. These are used to control which algorithms are to be executed when drawing the schematic. This is done to stimulate the user to try out different combinations of algorithms in order to select the best result according to his or her view. These settables are:

- `PLACEMENT_METHOD` - 1 for backward traversal, 0 for bubble sort.
- `HORIZONTAL_ALIGN` - The horizontal alignment algorithm as described in Section 4.6.1. 1 when enabled, 0 when disabled.
- `FANIN_CENTERING` - The Fanin Centering algorithm as described in Section 4.6.1. 1 when enabled, 0 when disabled.
- `FIX_CONNECTION_ROUTING` - if set to 1, the fixed geometrical routing algorithm for stem and branch type connections (see Section 5.2.4) is used, instead of the original routing. If set to 0, the original routing algorithm is used.
- `FIX_TRACK_ASSIGNMENTS` - if set to 1, the fixed track assignment algorithm (see Section 5.2.2) is used, instead of the original routing. If set to 0, the original track assignment algorithm is used.
- `PLADJUSTMENT` - 1 when switching the circuit's input pins is allowed, 0 when disabled.
- `PO_ADJUSTMENT` - 1 when switching the circuit's output pins is allowed, 0 when disabled.
- `PAIRWISE_SWAP` - The Pairwise Swap algorithm as described in Section 5.3. 1 when enabled, 0 when disabled.

In addition, the following settable scalars have been added to control drawing aspects of the schematic:

- `DEFAULT_CONE_COLOR` - the default color number for drawing cones
- `MIN_CONN_DIST` - the minimal distance in pixels between connections. Used to control routing.
- `MIN_CHANNEL_WIDTH` - the minimal number of tracks in the channel.

Conclusions and Further Research

6

The literature study which was part of the assignment showed a limited amount of usable procedures. The most promising procedures were either not explained in detail or already present in the Automatic Schematic Generator in some form or another. Section 3.14 presents the conclusions of the reviewed literature. A useful procedure which has been added is the Bubble Sort placement procedure (see [YJ91]). A second addition is the Pairwise Swap procedure. Both are explained in detail in Section 5.3.

Studying the existing implementation showed that assumptions were made which are hard coded throughout the program: before they can be changed, the whole program must be examined carefully to assess the consequences of these changes. One of these assumptions is that elements and channels occupy a fixed amount of space on the drawing surface of the schematic; this greatly simplified the code for final placement of the ordered components and connections on the drawing surface. The disadvantage is that it has become hard to change parts of the code without breaking one of the implicit program assumptions, creating error in unexpected parts of the ASG program. Therefore, the original structure and assumptions have been left intact and the improved and added procedures work with these assumptions.

During the study of the existing implementation, a number of bugs were found in the original ASG implementation. The circuit's input pins ordering procedure did not work correctly. The circuit output ordering procedure failed to notice empty slots in the circuit output column. The track assignment procedure assigned connections to tracks in a wrong sequence causing them to partly overlap each other. The connection routing procedure was designed with black and white drawings in mind. When coloring was added later, it showed that all branches

of a fanout point were routed from the fanout point, causing them to partly overlap each other. Coloring also showed that the stem of the fanout was drawn incorrectly. Line segments which were part of the stem connection were incorrectly shown as part of branch connections (ghost line problem). A complete list of these bugs and shortcomings can be found in Section 4.7.

These bugs, as well as some smaller ones, have all been solved as is described in Chapter 5.

In order to provide the user with more control over the ASG the relevant ASG procedures have been made switchable. The user can now choose which set of procedures should be executed. This stimulates the user to try different combinations of procedures to find the optimal schematic diagram for the circuit.

Another addition is the option to highlight different types of structure in the schematic: input cones, output cones, fanout free regions and element regions can be given different colors than the rest of schematic. The details of this are explained in Section 5.4. They have been added in the form of commands for DAT's command interface, as explained in Section 5.5.

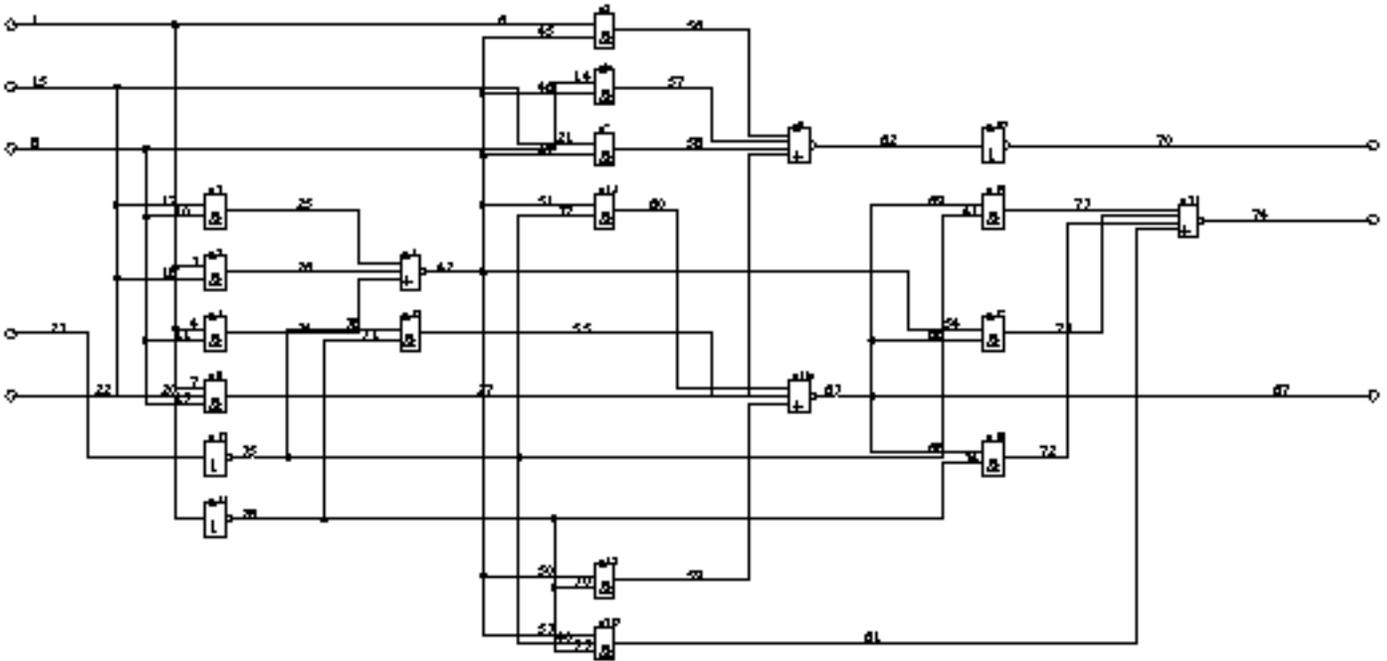


Figure 6.1: Circuit 7482 drawn with the original DAT-ASG.

This combination of fixing bugs in the existing implementation, improving existing procedures and adding new ones has improved the graphic output considerably. Figure 6.1 shows the 7482 circuit as drawn by the original DAT-ASG. It shows a number of bugs and shortcomings which have been fixed by the current implementation. They are presented in close-ups:

- Figure 6.2 shows the incorrect position assignment of the circuit's input pins. The two input pins at the left edge of the schematic can easily be drawn at the same height as their predecessors.

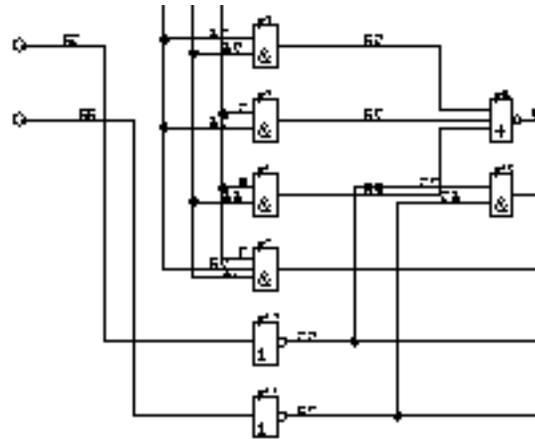


Figure 6.2: Circuit Inputs in circuit 7482 drawn with the original DAT-ASG

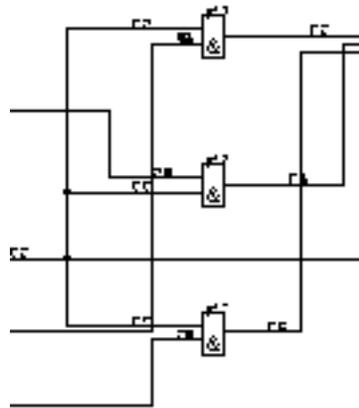


Figure 6.3: Incorrect Fanin Centering in circuit 7482 drawn with the original DAT-ASG.

- Figure 6.3 shows the incorrect fanin centering algorithm in the original implementation. The logic gate at the top should be drawn one position lower (there is space available for this), bringing it closer to the average row position of its predecessors.
- Figure 6.4 shows the horizontal overlap present in the original DAT-ASG. The connections to the left of the gate in the middle are partly overlapping each other.
- Figure 6.5 shows that the two middle elements in the column should switch positions to increase readability. They have predecessor elements at each other's row position, and are swapped by the Pairwise Swap procedure.

The same circuit is drawn with the improved DAT-ASG in figure 6.6 and figure 6.7. The horizontal overlap problem has been solved, making it possible to determine which signal lines belong to which connections. The circuit's input pins are now positioned directly in front of the elements they feed, reducing the number of signal line crossovers, bends, and total signal line length. The new fanin centering algorithm places elements closer to their predecessors, thereby decreasing the chance of signal line crossings through the reduction of total signal line

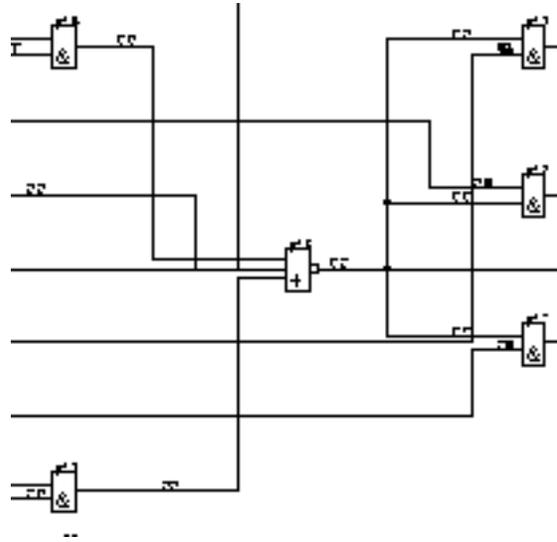


Figure 6.4: Horizontal Overlap in circuit 7482 drawn with the original DAT-ASG

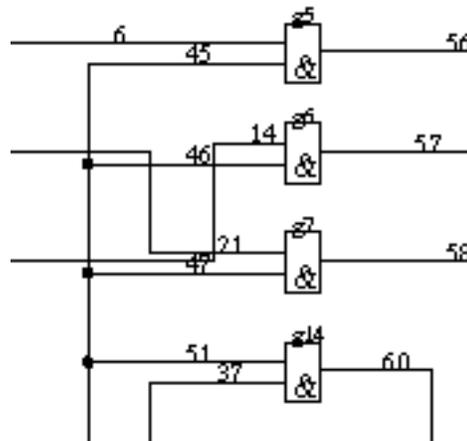


Figure 6.5: Pairwise twisted gates in circuit 7482 drawn with the original DAT-ASG

length. The pairwise swap algorithm has reduced both the number of crossovers and the number of signal line bends. Concluding, compared to the original situation the number of signal line crossovers, bends in signal line, and total signal line length has been reduced, thereby increasing the readability of the schematic.

The option to enable or disable individual procedures has also demonstrated what was already clear from the reviewed literature: there is no single best solution. Sometimes a procedure such as fanin centering reduces the readability. Even within a single schematic a procedure can improve one region of the schematic while worsening another. There is also no single optimization criteria for defining the readability of the schematic.

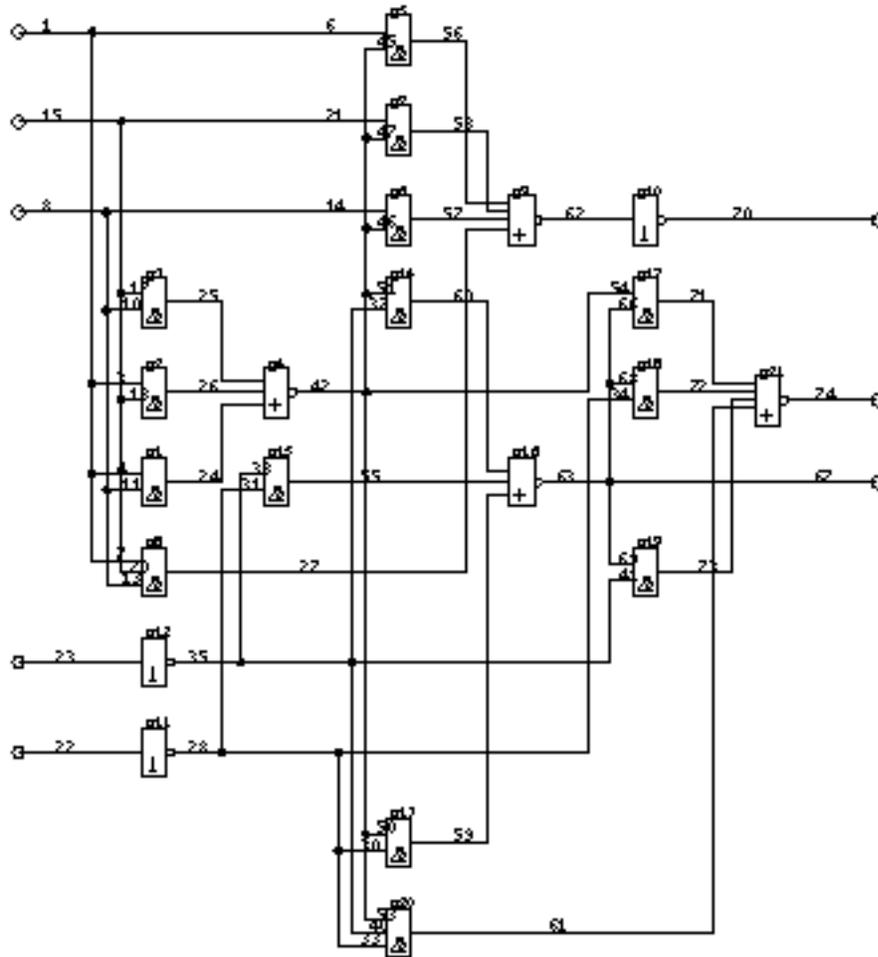


Figure 6.6: Circuit 7482 displayed by current DAT-ASG, using Backward Traversal for initial placement.

Further Research Further research into this topic should include a new track assignment procedure. The modified Left Edge routing method present in the current DAT-ASG tries to minimize the amount of tracks used for routing. While this is desirable in VLSI layout techniques, it is of lesser importance in ASG. The emphasis should be on the minimization of connection crossover. Looking at the schematics drawn with the current version of the DAT-ASG shows a high number of signal line crossovers which can be prevented when connections are assigned to different tracks. By implementing a completely new track assignment method, the number of signal line crossings can be reduced without changing the positions of the elements, thereby increasing the readability of the resulting schematic diagram.

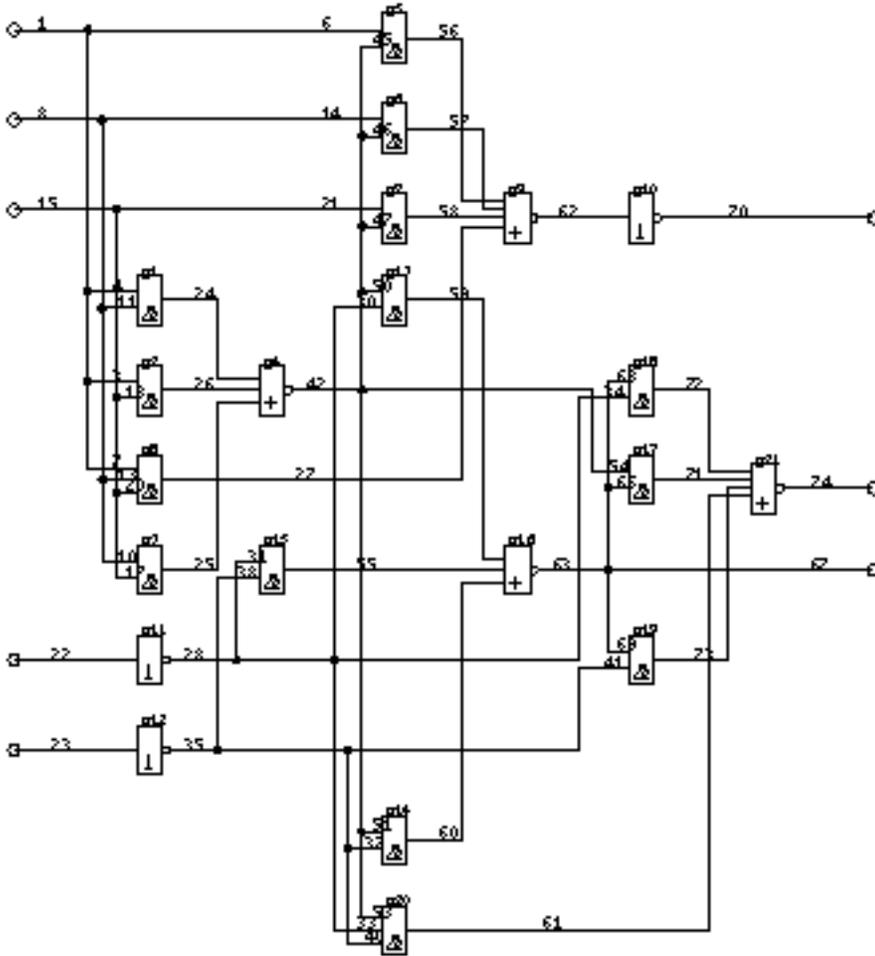


Figure 6.7: Circuit 7482 displayed by current DAT-ASG, using Bubble Sort for initial placement.

Bibliography

- [AKM86] V. Swaminathan A. Kumar, A. Arya and A. Misra. Automatic Generation of Digital System Schematic Diagrams. In *IEEE Design and Test of Computers*, pages 58–65, 1986.
- [BN93] K. Raughunathan B. Naveen. An Automatic Netlist-to-Schematic Generator. In *IEEE Design and Test of Computers*, pages 36–40, 1993.
- [Bre75] R.J. Brennan. An Algorithm for Automatic Line Routing on Schematic Diagrams. In *Proceedings of the 12 Design Automation Conference*, pages 324–330, 1975.
- [GS90] L. Hafer G.M. Swinkels. Schematic Generation with an Expert System. In *IEEE Transactions on Computer Aided Design, Vol. CAD-9 NO. 12*, pages 1289–1306, 1990.
- [LS89] G.J.P. Koster L. Stok. From Network to Artwork. In *Proceedings of the 26rd Design Automation Conference*, pages 686–689, 1989.
- [MA83] G.R. Stroick M.L. Ahlstrom, G.D. Hadden. Hal: A Heuristic Approach to Schematic Generation . In *IEEE International Conference on Computer Aided Design*, pages 83–86, 1983.
- [May85] M. May. Computer Generated Multi-Row Schematics. In *IEEE Transactions on Computer Aided Design, Vol. CAD-17 No. 1*, pages 25–29, 1985.
- [MM83] P. Mennecke M. May, A. Iwainsky. Placement and Routing for Logic Schematics. In *IEEE Transactions on Computer Aided Design, Vol. CAD-15 NO. 3*, pages 89–101, 1983.

- [MMA86] T.E. Fuhrman M.A. Majewski, F.N. Krull and P.J. Ainslie. AutoDraft: Automatic Synthesis of Circuit Diagrams. In *IEEE International Conference on Computer Aided Design*, pages 435–438, 1986.
- [RC87] L.P. McNamee R.K. Chun, K.J. Chang. Vision: VHDL Induce Schematic Imaging on Netlists. In *Proceedings of the 24rd Design Automation Conference*, pages 436–442, 1987.
- [Sab94] E.A. Sabbah. Automatic Generation of Logic Schematics from Netlist Description. Technical report, Delft Univ. of Technology, CARDIT Laboratory, 1994.
- [TL89] L.P. McNamee T.D. Lee. Structure Optimization in Logic Schematic Generation. In *IEEE International Conference on Computer Aided Design*, pages 330–333, 1989.
- [TY82] E.S. Kuh T. Yoshimura. Efficient Algorithms for Channel Routing. In *IEEE Transactions on Computer Aided Design, Vol. CAD-1 No. 1*, pages 25–35, 1982.
- [VV86] C.D. Wilcox V.V. Venkataraman. Gems: An Automatic Layout Tool for Mimola Schematics. In *Proceedings of the 23rd Design Automation Conference*, pages 131–137, 1986.
- [YJ91] T.M. Parng Y.S. Jehng, L.G. Chen. ASG: Automatic Schematic Generation. In *Integrated VLSI Journal, Vol. 11 No. 1*, pages 11–27, 1991.

ASG related Commands and Settables

This appendix give an overview of all Automatic Schematic Generator (ASG) related commands and settables in DAT's Command Line Interface. Note that:

- Typing *help* will give an overview of all commands available in DAT.
- Typing *help < commandname >* will give help for the specified command.
- Typing *set* will give an overview of all settables.
- Typing *sethelp < settable >* will give help for the specified settable.

A.1 ASG related Commands

circread

usage: *circread < circuitpath > [format < design > [librarypath]]*

Command CIRCREAD reads in a specified circuit. If successfully read the circuit, various preprocesses are run on it, and main DAT components are made selectable so commands like ATPG can be used. The optional flag FORMAT may be any of DAP, NDL, DAT. Default is DAP format. If NDL format is used, the DESIGN name must be given. If NDL format is used, a library may normally have to be specified, unless all information is already in the file containing the design. A library is specified by filename. DAT format is also writable. For now circuit may only be read in once.

draw

usage: *draw*[*psfile*]

Draws a picture of the circuit in specified file or the default file default.dat.ps. Output file will be in postscript format.

quit

usage: *quit*[*ALL*]

When DAT is in interactive command mode, QUIT exits DAT completely.

set

usage: *set* < *SettableScalar* >< *value* >

command SET is used to set Settable Scalars. If value is omitted, value is listed out. If Settable Scalar is also omitted, settables are listed with their current value.

xdraw

usage: *xdraw*

Invokes the ASG algorithms and draws a picture of the circuit in an X window.

xdrawconn

usage: *xdrawconn* < *connr* > [*colnr*]

Redraws a connection in the X display, in current given color.

xdrawconnlabel

usage: *xdrawconnlabel* < *connr* >< *mssg* >

Draws a connection label in the X display, in current color.

xdrawffr

usage: *xdrawffr* < *connr* >< *levels* > [*colnr*]

Set the drawing color of a Fanout Free Region which starts at connection ;*connr*;

xdrawinputcone

usage: *xdrawinputcone* < *connr* >< *levels* > [*colnr*]

Set the drawing color of an input cone which is ;*levels* levels deep and start at connection ;*connr*;

xdrawoutputcone

usage: *xdrawoutputcone* < *connr* >< *levels* > [*colnr*]

Set the drawing color of an output cone which is ;*levels* levels deep and start at connection ;*connr*;

xdrawregion

usage: *xdrawregion* < *connr* >< *nback* >< *nforward* > [*colnr*]

Set the drawing color of a region cone which goes ;*nback* levels back and ;*nforward* levels forward and start at connection ;*connr*;

xdrawresetconncolors

usage: *xdrawresetconncolors*[*colornr*]

Set the drawing color of each connection to the default color (0) or to [*colornr*].

xdrawsetcolor

usage: *xdrawsetcolor* < *colnr* >

Sets the current color for the X display.

xdrawsetmssg

usage: *xdrawsetmssg* < *mssg* >

Redraws a message in the X message display, in current color.

xdrawsettitle

usage: *xdrawsettitle* < *mssg* >

Redraws a title in the X title display, in current color.

xdrawunsetmssg

usage: *xdrawunsetmssg*

Removes (pops) a message from the X message display.

xdrawunsettitle

usage: *xdrawunsettitle*

Removes (pops) a title from the X title display.

xdrawupdate

usage: *xdrawupdate*

Update X display. Automatic updates are done if UPDATEXDRAW is set.

A.2 ASG related Settables

CIRCUITPATH

set *CIRCUITPATH* < *circuitdirectorypath* >

Sets default path for circuit files.

DISPLAYONCE

set *DISPLAYONCE* < 0 – 1 >

Display continuously or once in command *xdraw*.

DRAWLANDSCAPE

set *DRAWLANDSCAPE* < 0 – 1 >

If set to 1, postscript prints are done in landscape orientation.

FANIN_CENTERING

set *FANIN_CENTERING* < 0 – 1 >

If set to 1, the Fanin Centering algorithm is enabled, if set to 0 it is disabled.

FIX_CONNECTION_ROUTING

set **FIX_CONNECTION_ROUTING** < 0 – 1 >

If set to 1, the Fix Connection Routing algorithm is enabled, if set to 0 it is disabled.

FIX_TRACK_ASSIGNMENTS

set **FIX_TRACK_ASSIGNMENTS** < 0 – 1 >

If set to 1, the Fix Track Assignments algorithm is enabled, if set to 0 it is disabled.

HORIZONTAL_ALIGN

set **HORIZONTAL_ALIGN** < 0 – 1 >

If set to 1, the Horizontal Alignment algorithm is enabled, if set to 0 it is disabled.

PAIRWISE_SWAP

set **PAIRWISE_SWAP** < 0 – 1 >

If set to 1, the Pairwise Swap algorithm is enabled, if set to 0 it is disabled.

PI_ADJUSTMENT

set **PI_ADJUSTMENT** < 0 – 1 >

If set to 1, the PI Adjustment (circuit input switching) algorithm is enabled, if set to 0 it is disabled.

PLACEMENT_METHOD

set **PLACEMENT_METHOD** < 0 – 1 >

Selects the initial placement algorithm used by the ASG. 0 for BubbleSort, 1 for Backwards Traversal.

PO_ADJUSTMENT

set **PO_ADJUSTMENT** < 0 – 1 >

If set to 1, the PO Adjustment (circuit output switching) algorithm is enabled, if set to 0 it is disabled.

PSMULTIPAGE

set **PSMULTIPAGE** < 0 – 1 >

If set to 1, ps prints are done in multiple pages if needed.

PSNONENCAPSULATED

set **PSNONENCAPSULATED** < 0 – 1 >

If set to 1, postscript prints are done in non-encapsulated format.

PSSCALEDIV

set **PSSCALEDIV** < 1 – *MAXUNSIGNED* >

Divisor for determining scaling factor in postscript or X display.

PSSCALEMUL

set PSSCALEMUL < 1 – MAXUNSIGNED >

Multiplier for determining scaling factor in postscript or X display.

UPDATEXDRAW

set UPDATEXDRAW < 0 – 1 >

If set, each xdrawconn etc is immediately updated.